Univerza v Ljubljani
Fakulteta za računalništvo in informatiko

Luka Pirnat

Načrtovanje vgrajenih sistemov na čipu Xilinx Zynq

diplomsko delo

univerzitetni študijski program
prve stopnje
računalništvo in informatika

Mentor: izr. prof. dr. Patricio Bulić

Ljubljana, 2016

Besedilo je oblikovano z urejevalnikom besedil LaTeX.

Fakulteta za računalništvo in informatiko izdaja naslednjo nalogo:

Načrtovanje vgrajenih sistemov na čipu Xilinx Zynq

Tematika naloge:

Preučite načrtovanje vgrajenih sistemov v programabilnih čipih Xilinx Zynq. Nato v čipu načrtujte in sprogramirajte sistem za obdelavo zvoka, ki bo uporabljal vgrajeno ARM jedro, notranje sistemsko vodilo, pomnilnik z DMA krmilnikom, FFT blokom iz knjižnice IP blokov ter enim lastno načrtovanim blokom v FPGA. Programsko opremo načrtujte na najnižjem nivoju brez uporabe ponujenega operacijskega sistema z gonilniki (t.i. bare-metal programming).

Rad bi se zahvalil predragi Ani za vso pomoč v zadnjem mesecu in spodbudo v trenutkih brezupa in staršem za podporo in omogočanje študija.
Zahvaljujem se tudi prof. Buliću za pomoč in prilagodljivost.

Kazalo

Seznam uporabljenih kratic

kratica	angleško	slovensko
FPGA	Field Programmable Gate Array	programabilno logično vezje
SoC	System-on-Chip	sistem na čipu
PL	Programmable Logic	programabilna logika
PS	Processing System	procesorski sistem
AXI	Advanced eXtensible Interface	komunikacijski vmesnik AXI
DSP	Digital Signal Processing	digitalno procesiranje signalov
HDL	Hardware Description Language	jezik za opis vezij
VHDL	VHSIC Hardware Description Language	jezik za opis vezij VHSIC
FFT	Fast Fourier Transform	hitra Fourierova transformacija
IP	Intellectual Property	logični blok IP
RAM	Random-Access Memory	bralno-pisalni pomnilnik
ROM	Read-Only Memory	bralni pomnilnik
CLB	Configurable Logic Block	konfigurabilni logični blok
LUT	Lookup Table	logična tabela
I2S	Integrated Interchip Sound	standard za prenos zvoka med vezji

Povzetek

Naslov: Načrtovanje vgrajenih sistemov na čipu Xilinx Zynq

Avtor: Luka Pirnat

V delu predstavimo programabilni sistem na čipu Xilinx Zynq in delo z njim. Pregledamo področja, na katerih se ga zaradi svojih značilnosti uporablja in opišemo motivacijo za njegovo izbiro pri implementaciji vgrajenih sistemov. Primerjamo ga z drugimi čipi. Natančneje opišemo njegovo dvodelno zgradbo in njegove posamezne gradnike v vsakem delu posebej. Nato pokažemo, kako jih uporabljati na primeru načrtovanja vgrajenega sistema, ki zajema zvok prek mikrofona ali linijskega vhoda, določi njegovo frekvenco in zna predvajati sintetizirane kitarske tone. Sistem implementiramo in skozi opis njegovega načrtovanja in implementacije prikažemo različne vidike dela s čipom.

Ključne besede: sistem na čipu, Xilinx Zynq, FPGA, AXI, Vivado.

Abstract

Title: Embedded System Design on Xilinx Zynq

Author: Luka Pirnat

In the thesis we introduce the concept of programmable System-on-Chip (SoC) and continue with a representative SoC device Xilinx Zynq. We compare it with other FPGA and embedded systems devices and introduce its pros. Next we describe its two-part structure and list the elements of each of the two parts. Then we describe the more important ones a bit more precisely. In the second part we show how to design an embedded system on the chip and describe tools and techniques used.

Keywords: System-on-Chip, Xilinx Zynq, FPGA, AXI, Vivado.

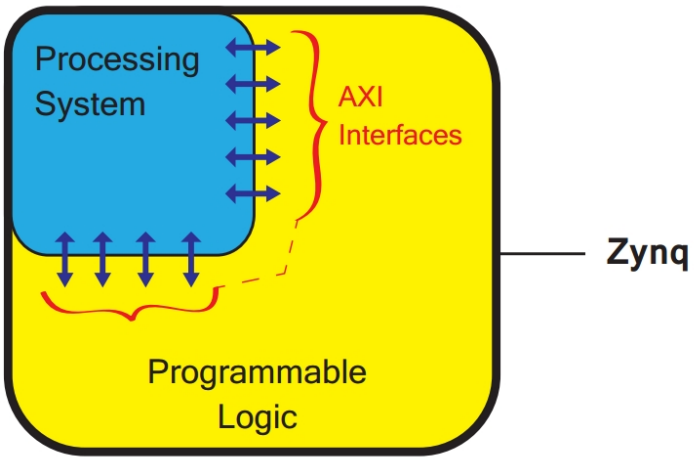
Poglavje 1

Uvod

Pri razvoju integriranih vezij že dlje časa obstaja težnja, da se čim več funkcionalnih enot oz. kar cel sistem implementira na istem vezju – čipu. Zaradi hitrega razvoja tehnologije polprevodnikov pa se to v zadnjem času tudi vse bolj uresničuje. Mnogokrat zmogljivejši računalniški sistemi od tistih v velikosti omare iz bližnje preteklosti, so danes na enem samem čipu. Kot primer integracije več funkcionalnih enot lahko navedemo severni most (*northbridge*) z matične plošče osebnega računalnika, ki se je preselil v centralno procesno enoto; ta v prenosnih računalnikih pogosto vsebuje tudi grafično procesno enoto. Še očitnejši primeri so čipi v modernih mobilnih telefonih, ki implementirajo celoten sistem – procesor, grafični procesor, pomnilnik, lahko celo radiofrekvenčni del (WiFi, 3G, 4G) pa čip na računalniku Raspberry Pi, ki ima večino svojega sistema, ki je enakovreden tistim osebnih računalnikov nižjega cenovnega razreda, na enem samem čipu [8].

Taki čipi se imenujejo *SoC – System-on-Chip*, saj, kot smo ugotovili, implementirajo celoten računalniški sistem. Prednosti sistemov na čipu so nižja cena izdelave, hitrejši in varnejši prenos podatkov med različnimi deli sistema, manjša poraba energije, majhnost in večja zanesljivost. Vendar pa imajo tudi ti sistemi svoje omejitve, sploh kadar želimo čim krajši čas razvoja, fleksibilnost in možnost naknadne nadgradnje [3]. Zato obstajajo tudi „sistemi na programabilnih čipih“, ki so bila do zdaj vezja FPGA – v celoti

reprogramabilna vezja. Nekaj let nazaj pa so se pojavili sistemi, sestavljeni tako iz programabilnega vezja FPGA kot procesorja s perifernimi napravami. Njihova prednost je v večji hitrosti prenosa podatkov med vezjem FPGA in procesorjem zaradi dejstva, da sta oba na istem čipu. To je pomembno zato, ker je v vgrajenih sistemih običajno, da je v njih vsaj en procesor, ki upravlja celoten sistem, gosti programe in upravlja s perifernimi napravami. Če imamo torej navaden FPGA, moramo procesor ali dodati na skupno tiskano vezje ali pa ga implementirati na vezju FPGA kot t.i. *mehki procesor*.
V skupino programabilnih sistemov SoC spada tudi subjekt pričujočega dela – Xilinx Zynq, ki je torej iz dveh delov – vezja FPGA oz. programabilne logike (PL) in procesorskega sistema (PS), kar je predstavljeno na sliki 1.1.



Slika 1.1: Poenostavljen model arhitekture sistema na čipu Zynq [3].

Programabilna logika je v celoti programabilno vezje, na katerem lahko implementiramo poljubno napravo za komunikacijo z zunanjostjo ali pa koprocesor za hitrejše procesiranje, procesorski sistem pa je sistem z dvojedrnim procesorjem ARM Cortex-A9 in množico perifernih naprav. Prvo je torej idealno za hiter pretok podatkov ali hitro procesiranje, drugo pa za izvajanje programov, lahko tudi operacijskega sistema. Komunikacija med obema deloma poteka prek komunikacijskih vmesnikov industrijskega standarda *AXI*

– *Advanced eXtensible Interface* [3].

Sistem na čipu Zynq bomo predstavili podrobneje in na primeru razvoja aplikacije za digitalno procesiranje zvoka prikazali, kako deluje in kako se na njem implementira lastno aplikacijo.

Poglavje 2

Predstavitev sistema na čipu Zynq

V nadaljevanju bomo natančneje predstavili zaokrožena dela sistema Zynq – programabilno logiko in procesorski sistem, komunikacijo med njima in znotraj njiju ter pomembnejše periferne naprave.

2.1 Programabilna logika (PL)

Zynqova programabilna logika je pravzaprav FPGA, enakovreden Xilinxovim čipom FPGA, in v sistem prinaša reprogramabilnost in hitrost zaradi strojnega paralelizma. To je glavna prednost sistema v primerjavi s sistemi s samim procesorjem brez FPGA. Pri manjših in cenejših sistemih Zynq je PL enakovredna čipom FPGA Artix-7, pri večjih pa Kintex-7 [3]. Pri velikosti imamo v mislih število funkcionalnih enot na čipu – konfigurabilnih logičnih enot, pomnilniških celic, blokov za digitalno procesiranje signalov itn.

Kot že omenjeno, lahko v PL implementiramo poljubno digitalno napravo, lahko tudi mehki procesor, ki kot koprocesor za manj zahtevne naloge služi jedroma ARM.

2.1.1 FPGA

Za boljše in celovitejše razumevanje bomo kratko predstavili tehnologijo *FPGA – Field Programmable Gate Array*.

Začetki tehnologije segajo v leto 1970, ko so pri Texas Instruments izdelali prvo programabilno vezje, ki je temeljilo na pomnilniku ROM in se je sprogramiralo že pri izdelavi. Leta 1984 so v podjetju Altera naredili naslednji korak z reprogramabilnim vezjem – vezjem, ki se ga da ponovno sprogramirati pri uporabniku. To se je dalo storiti z ultravijolično lučjo, s katero je bilo treba posvetiti na čip skozi posebno kvarčno okno v njem. Prvo komercialno vezje FPGA v obliki, kot jo poznamo danes, pa so naredili leta 1985 v Xilinxu. Taka vezja so brez specialnih naprav programabilna pri uporabniku, od tod izraz *field-programmable* [7].

Na čipu FPGA lahko implementiramo poljubno kombinatorično funkcijo, pri čemer je njena kompleksnost omejena s številom razpoložljivih logičnih blokov na čipu. Pri implementaciji na FPGA pa lahko dosežemo veliko večjo hitrost izvajanja v primerjavi s procesorjem, saj se procesi izvajajo vzporedno. Programiramo v jeziku *HDL – Hardware Description Language*, ki je opis digitalnega vezja, ki ga želimo implementirati. Najbolj znana jezika HDL sta VHDL in Verilog.

Zaradi prve lastnosti – možnosti implementacije poljubnega digitalnega vezja – se FPGA-ji uporabljajo za načrtovanje vezij, saj skrajšajo čas, ker vezja med načrtovanjem in testiranjem ni treba fizično izdelati. Zaradi druge lastnosti pa se uporabljajo tam, kjer sicer cenejši procesorji niso dovolj zmogljivi, v vojaški industriji, letalstvu, zdravstvu (MRI, rentgen itd.), omrežnih napravah oz. v splošnem pri digitalnem procesiranju signalov na najrazličnejših področjih.

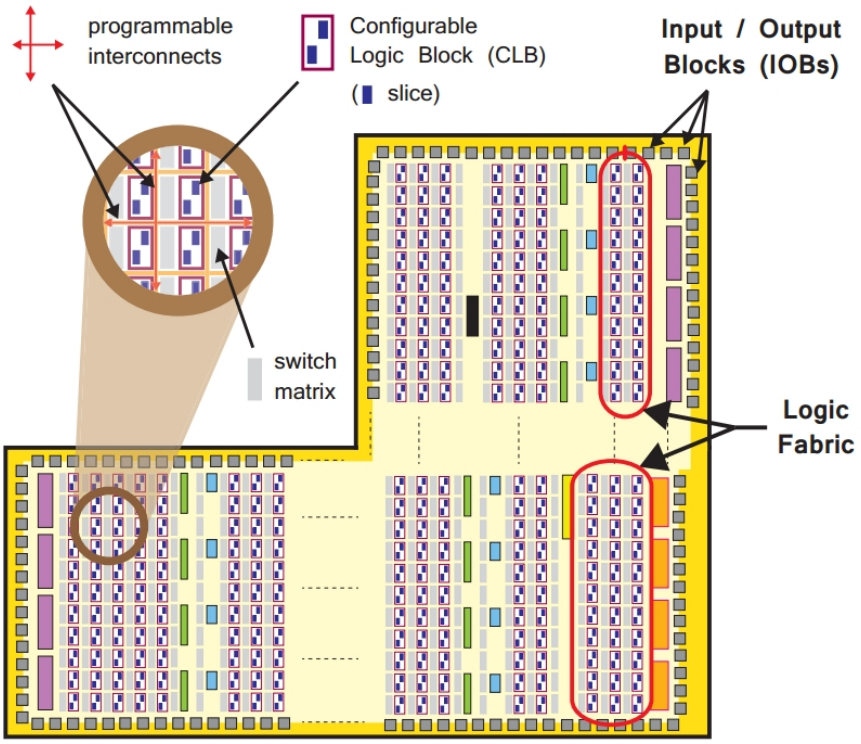
Zgradba FPGA

FPGA je zgrajen iz:

- **logičnega dela** (*logic fabric*), ki je njegov glavni gradnik in je dalje zgrajen iz:
  - **rezin** – enot, v katerih se implementira želeno logično vezje. Vsebujejo:
    - 4 **lookup tabele (LUT-e)**, ki dejansko implementirajo logično funkcijo z do 6 vhodi, ROM, RAM ali pa pomikalni register
    - 8 **zapahov** – **flip-flopov**, ki tvorijo registre; so torej pomnilne celice
  - **konfigurabilnih logičnih blokov (CLB-jev)**, ki so majhna 2-dimenzionalna polja logičnih elementov in vsebujejo po dve rezini
- **vhodno-izhodnih blokov (IOB-jev)** za komunikacijo z zunanjostjo
- **preklopnih matrik** in **programabilnih povezav** za povezovanje med posameznimi elementi
- **rezin DSP48E1**, namenjenih digitalnemu procesiranju signalov
- **bločnih RAM-ov** za hrambo večje količine podatkov
- **blokov XADC** za pretvorbo analognih signalov v digitalne
- **urinih blokov** za tvorjenje urinih signalov

Zgradba je jasno predstavljena na sliki 2.1.

Logične funkcije so torej implementirane s 6-vhodnimi LUT-i, ki imajo le en izhod. Če želimo implementirati večje funkcije, se več LUT-ov poveže med sabo. Enostaven primer funkcije AND je predstavljen s tabelo 2.1 na 2-vhodnem LUT-u, izbranem zaradi enostavnosti (6-vhodni LUT ima 2^6 različnih možnih vhodov). Vidimo, da je izhod 1 le, ko sta oba vhoda enaka 1.



Slika 2.1: Na sliki so označeni gradniki sistema FPGA, pri čemer so zeleni bločni RAM-i, modri rezine DSP48E1, črn je blok XADC, bloki za tvorjenje ure pa vijolični [3].

Take vrednosti bi se v praksi zapisale v LUT, ki pravzaprav deluje kot ROM, pri čemer vhodi predstavljajo naslove, izhod pa prebran podatek.

a	b	y
0	0	0
0	1	0
1	0	0
1	1	1

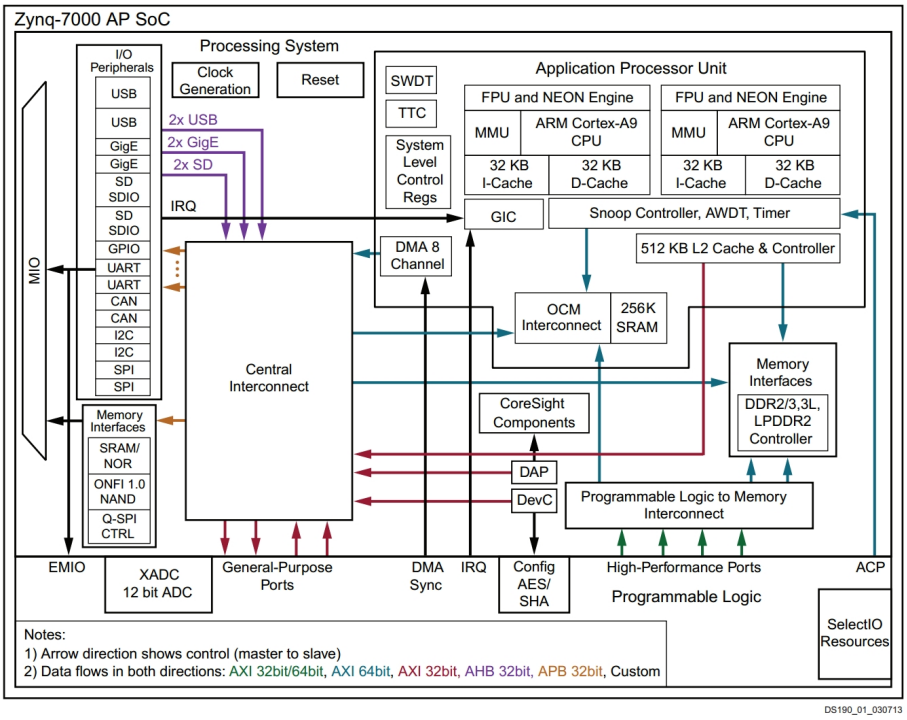
Tabela 2.1: Funkcija AND v 2-vhodnem LUT-u.

2.2 Procesorski sistem (PS)

Osnovni gradnik procesorskega sistema je dvojedrni procesor ARM Cortex-A9, ki lahko deluje na frekvencah do 1GHz in skupaj s celotnim PS predstavlja glavno prednost v primerjavi z navadnimi čipi FPGA. Na njih sicer res lahko implementiramo mehki procesor ali pa FPGA povežemo z zunanjim trdim procesorjem, a ta dva načina ne moreta doseči visokih hitrosti prenosa podatkov in zanesljivosti sistema Zynq, ki je toliko boljši zaradi namenskega procesorja in tesne povezave s PL.

2.2.1 Zgradba

Procesorski sistem čipa Zynq ni sestavljen samo iz procesorja, njegova celotna zgradba je razvidna s slike 2.2.



Slika 2.2: Procesorski sistem čipa Zynq [19].

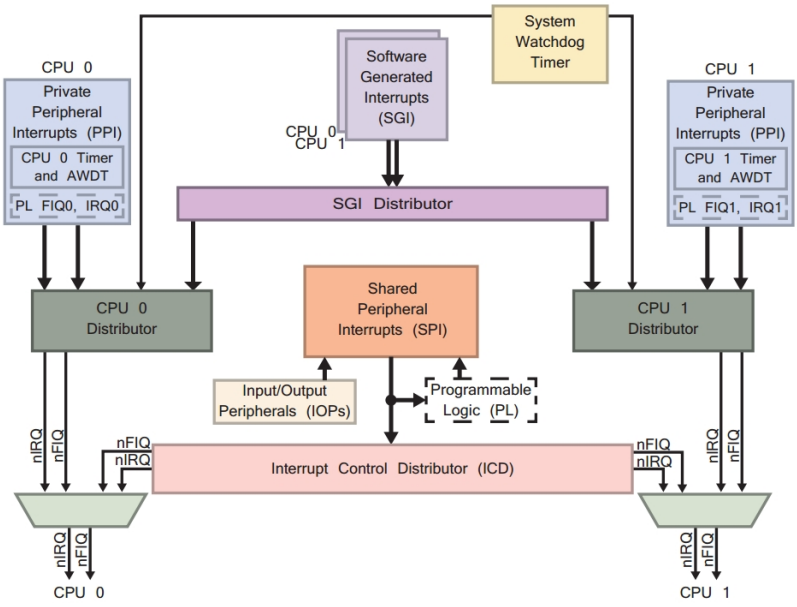
PS med drugim vsebuje:

- **aplikacijsko procesno enoto (APU)**, ki sestoji iz:
  - **procesorskih jeder ARM**
  - **predpomnilnikov**
  - **enote za upravljanje s pomnilnikom (MMU)**, ki preslikava med virtualnimi in fizičnimi naslovi
  - enote za računanje s **plavajočo vejico** skupaj z razširitvijo SIMD za digitalno procesiranje signalov
  - **pomnilnika OCM**
  - **enote SCU**, ki tvori most med jedri in L2-predpomnilnikom in skrbi za konsistentnost podatkov v predpomnilnikih
  - **časovnikov**
  - **prekinitvenega krmilnika**
- **pomnilniške vmesnike**
- **vhodno-izhodne periferne naprave**:
  - **multipleksiran vhod/izhod (MIO)**, ki ponuja 54 nastavljivih pinov za komunikacijo z zunanjostjo (ne s PL), kar pomeni, da lahko posamezne pine poljubno dodelimo perifernim napravam, npr. UART-u
  - **GPIO** za komunikacijo z enostavnimi napravami
  - vse periferne V/I-naprave so navedene v viru [19]
- **osrednji komunikacijski blok**

2.2.2 Generični prekinitveni krmilnik (GIC)

Generični prekinitveni krmilnik je centralni prekinitveni krmilnik v sistemu, zato sprejema vse prekinitve, jih razvršča po prioritetah in jih pošilja naprej enemu od jeder ARM ali pa v PL. Zmožen je tudi omogočiti ali onemogočiti posamezne vire prekinitev.

GIC je pomnilniško preslikan, procesorski jedri pa imata za dostop do njega vsako svoj privatni vmesnik, namenjen le dostopu do prekinitvenega krmilnika, kar zagotavlja hitrejšo odzivnost [3]. Na sliki 2.3 vidimo bločno shemo prekinitvenega krmilnika in možne izvore prekinitev.



Slika 2.3: Generični prekinitveni krmilnik [3].

Prekinitve so vektorske in imajo vsaka svojo identifikacijsko številko (ID), ki predstavlja indeks v tabeli naslovov prekinitveno-servisnih podprogramov (PSP). Te moramo za posamezne prekinitve napisati sami in njihov naslov zapisati na ustrezno mesto v tabelo naslovov. Naš PSP je klican prek servisnega podprograma posebne izjeme (*exception*), ki se sproži ob vseh prekinitvah IRQ (ne pa npr. ob FIQ). Pred dejanskim PSP se torej ob izjemi, ki se

zgodi zaradi prekinitve IRQ, kliče funkcija, ki kliče PSP, kar je vidno v kodi 2.1.

```
TablePtr = &(InstancePtr->Config->HandlerTable[InterruptID]);
if(TablePtr != NULL) {
    TablePtr->Handler(TablePtr->CallBackRef);
}
```

Izvorna koda 2.1: Klic našega PSP iz servisne rutine izjeme. Najprej preveri, ali je naslov nastavljen, in če je, kliče PSP.

Prekinitve, tja kamor so namenjene, posreduje ločen del krmilnika, imenovan prekinitveni distributer. Skrbi za razvrščanje prioritet, tipe sprožilcev prekinitev (fronta ali nivo) in posredovanje prekinitve ustreznemu jedru. Ko CPE prejme prekinitev, jo mora potrditi, distributer pa ji potem pošlje ID prekinitve. Ko CPE konča s procesiranjem, to sporoči, distributer pa spremeni status prekinitve iz aktivnega v neaktivnega.

Vrste prekinitev

Prekinitve v Zynqu delimo na tri skupine:

- **procesorjeve privatne periferne prekinitve (PPI)**, ki so povsem lastne posameznemu jedru CPE. Kot vidimo s slike 2.3, so to:
  - prekinitev **globalnega časovnika watchdog**, ki je skupen obema jedroma,
  - **hitra prekinitev iz PL (nFIQ)**,
  - prekinitev **privatnega časovnika**,
  - prekinitev **privatnega časovnika watchdog**
  - **prekinitev iz PL (nIRQ)**
- **deljene periferne prekinitve (SPI)**, ki jih je okrog 60 in so lahko namenjene enemu ali obema CPE-jema ali pa PL. Celoten seznam je na voljo v [3] in [19], na njem pa so med drugim prekinitve časovnika

iz PS pa prekinitve perifernih V/I-naprav USB, I2C, UART, Ethernet. Za nas so zanimive predvsem prekinitve iz PL v PS z ID-ji od 61 do 68 in od 84 do 91, ki se imenujejo **IRQF2P**, ki je sicer 20-bitno vodilo, a so zgornji 4 biti namenjeni prej omenjenim nFIQ in nIRQ za obe jedri.

- **programsko generirane prekinitve**, ki jih je 16, tvorita pa jih jedri. Ti lahko prekineta sami sebe ali pa drugo jedro

Postopek inicializacije GIC-a

V programu, ki ga napišemo, moramo poskrbeti za inicializacijo prekinitvenega krmilnika, napisati prekinitveno-servisni podprogram (PSP) in njegov naslov shraniti v tabelo naslovov PSP-jev, ki se hrani v GIC-evem objektu, ki ga moramo ustvariti ob inicializaciji. To nam je precej olajšano, saj od Xilinxa z namestitvijo razvojnega okolja dobimo tudi gonilnike za GIC. Postopek je prikazan z odseki kode 2.2, 2.3 in 2.4.

```
typedef struct
{
    u16 DeviceId;           // Unique ID  of device
    u32 CpuBaseAddress;     // CPU Interface Register base addr.
    u32 DistBaseAddress;    // Distributor Register base addr.
    //Vector table of interrupt handlers
    XScuGic_VectorTableEntry HandlerTable[XSCUGIC_MAX_NUM_INTRS];
} XScuGic_Config;
```

Izvorna koda 2.2: Struktura, ki hrani konfiguracijo krmilnika GIC – ID naprave in njen naslov (naprava je, kot smo omenili, pomnilniško preslikana), naslov distributerja in kazalec na tabelo naslovov.

```
cfg_ptr = XScuGic_LookupConfig(XPAR_PS7_SCUGIC_0_DEVICE_ID);
if (cfg_ptr)
    XScuGic_CfgInitialize(
        p_intc_inst,    // kazalec na instanco GIC
        cfg_ptr,
        cfg_ptr->CpuBaseAddress);
```

Izvorna koda 2.3: Inicializacija krmilnika. Najprej preberemo njegovo konfiguracijo, potem pa ga z njo inicializiramo. Funkcija tabelo z naslovi funkcij PSP nastavi na naslov prazne funkcije, ki se izvrši, če pozabimo nastaviti svoj PSP, zatem inicializira še distributerja in vmesnik med njim in jedroma CPE.

```
XScuGic_Connect(
    p_intc_inst,    // kazalec na instanco GIC-a
    ID_prekinitve,
    (Xil_InterruptHandler)PSP,
    p_dma_inst);    // argument za nas PSP, v tem primeru kazalec
                    // na vir prekinitve (DMA)

XScuGic_Enable(p_intc_inst, ID_prekinitve); // omogocimo prek.

// Vpis naslova funkcije v tabelo naslovov serv. podp. izjem
Xil_ExceptionRegisterHandler(
    XIL_EXCEPTION_ID_INT,   // ID izjeme za IRQ
    // servisni podprogram za izjeme (del firmwara)
    (Xil_ExceptionHandler)XScuGic_InterruptHandler,
    p_intc_inst);

// Omogocenje izjem IRQ
Xil_ExceptionEnable();
```

Izvorna koda 2.4: Treba je še vpisati naslov naše funkcije PSP v tabelo naslovov, omogočiti želeno prekinitev, potem pa naslov servisnega podprograma za izjeme vpisati v tabelo naslovov servisnih podprogramov izjem. Nazadnje je treba le še omogočiti izjemo za prekinitve IRQ.

2.3 Komunikacijski vmesniki med PL in PS

Da se v celoti izkoristi tesna povezanost obeh delov sistema je potreben učinkovit in zmogljiv komunikacijski protokol. V Zynqu je to četrta verzija standarda *AXI* oz. *Advanced eXtensible Interface* AXI4 [3].

2.3.1 Standard AXI4

AXI je del standarda AMBA, ki so ga leta 1996 razvili v podjetju ARM za komunikacijo v mikrokontrolerjih. Z leti je postal de-facto standard za komunikacijo znotraj čipov, pri Zynqu se tako uporablja za komunikacijo tako med PL in PS kot znotraj obeh delov sistema [3]. Obstajajo tri različice standarda AXI4, vsaka namenjena drugačnim zahtevam:

- **AXI4** – za pomnilniško preslikane povezave s potrebo po največji zmogljivosti. Gre za eksplozivni prenos; ko se na vodilo postavi naslov, mu sledi do 256 podatkovnih besed.
- **AXI4-Lite** – za enostavne počasne pomnilniško preslikane povezave. Po naslovu sledi največ ena podatkovna beseda.
- **AXI4-Stream** – za hitre tokove podatkov, ko povezave niso pomnilniško preslikane – ni mehanizma naslavljanja. Uporablja se npr. pri prenašanju tokov video podatkov, zvoka ipd. Realiziran je z najmanj signali od vseh treh. Ti signali so [1][16]:
  - **TVALID** je edini obvezni signal, njegov izvor je gospodar, sužnju pa pove, ali gospodar pošilja veljavne podatke.
  - **TREADY** je zelo priporočljiv, njegov izvor je suženj, gospodarju pa pove, ali je suženj pripravljen na sprejem podatkov. TVALID in TREADY sta signala, ki izvedeta rokovanje, prenos se lahko začne, ko sta oba postavljena na 1.
  - **TDATA** je podatkovno vodilo, njegov izvor je gospodar.

- **TSTRB** je osemkrat krajši od TDATA, njegova dolžina je torej število bajtov v TDATA, njegov izvor je gospodar, sužnju pa pove, kateri bajti so podatki in kateri pozicijski bajti. Ni pogosto uporabljen.
- **TKEEP** je iste dolžine kot TSTRB, njegov izvor je gospodar, sužnju pa pove, kateri bajti so veljavni podatki. Ni pogosto uporabljen.
- **TLAST** označuje konec paketa, njegov izvor je torej gospodar.

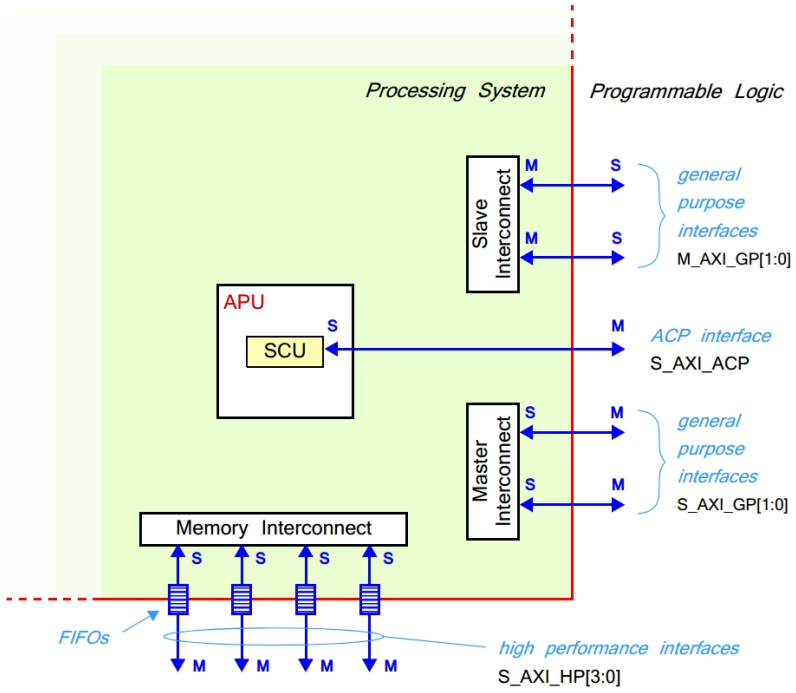
Če je protokol pomnilniško preslikan, to pomeni, da pri prenosu (branju ali pisanju) gospodar poda naslov, s katerega želi brati oz. pisati. Naslov je iz sistemskega pomnilniškega prostora.

2.3.2 Vmesniki in nadzorne komunikacijske enote standarda AXI

Glavna komunikacija med PL in PS poteka prek devetih vmesnikov AXI, ki so sestavljeni iz več kanalov. Dva pomembna elementa v komunikacijskem kanalu sta:

- **nadzorna komunikacijska enota (Interconnect)**, ki skrbi za preklapljanje prometa med različnimi vmesniki AXI. Znotraj PS je več teh enot, ki so namenjene bodisi interni uporabi bodisi za komunikacijo med PS in PL.
- **vmesnik** se uporablja za povezavo točka–točka med gospodarjem in sužnjem, prek katere tečejo podatki, naslovi in pomožni signali.

Vsi vmesniki so povezani z nadzornimi komunikacijskimi enotami znotraj PS, razen vmesnika ACP, ki je povezan s SCU-jem, ki smo ga že omenjali, ko smo opisovali aplikacijsko procesno enoto. Opisano je razvidno s slike 2.4.



Slika 2.4: Na sliki so prikazane vse povezave AXI med PS in PL [3].

Zynq pozna tri različne vmesnike AXI, ki so tudi označeni na sliki 2.4:

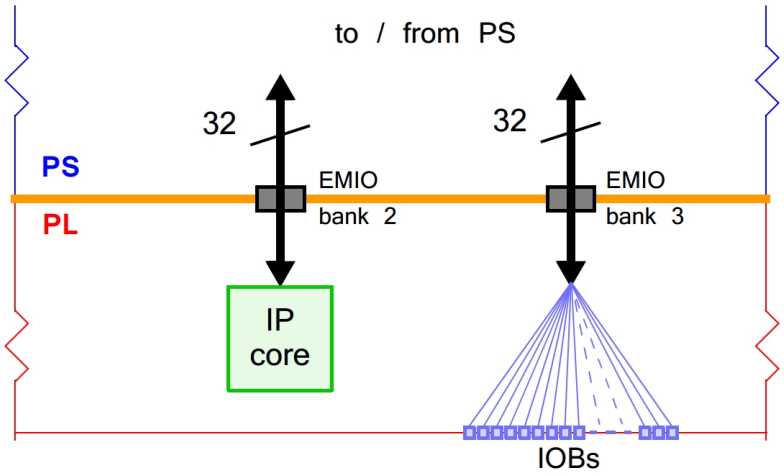
- **splošnonamenski vmesnik (AXI_GP)** je 32-bitni vmesnik, ki se uporablja za počasne in srednje hitre povezave brez medpomnjenja. So štirje, dva, v katerih je gospodar PS, in dva, v katerih je gospodar PL
- **vmesnik ACP (AXI_ACP)** je 64-bitna povezava med PL in enoto SCU, v kateri je PL gospodar, uporablja pa se za zagotavljanje konsistentosti podatkov med predpomnilniki in PL
- **visokozmogljivi vmesnik (AXI_HP)**, ki je lahko 32 ali 64-biten, je namenjen najhitrejšim povezavam in uporablja medpomnjenje. PL je gospodar.

Za realizacijo teh treh vmesnikov z vsemi njihovimi kanali, ki jih sestavljajo podatkovni, naslovni in kontrolni signali je potrebnih več kot 1000 signalov.

2.3.3 Razširjeni MIO in drugi povezovalni signali

Povezave med PS in zunanjostjo se lahko realizira tudi preko pinov PL, in sicer prek razširjenega MIO, **EMIO**. S tem pridobimo še več možnosti za povezavo z zunanjostjo.

EMIO pa se lahko uporabi tudi za povezavo PS in bloka v PL. Obe možnosti sta prikazani na sliki 2.5.



Slika 2.5: Uporaba vmesnika EMIO za povezavo poljubnega jedra v PL in procesorskega sistema (na levi) oz. poljubnega pina PL in procesorskega sistema (na desni) [3].

Med PS in PL obstaja še nekaj drugih signalov, med drugim za dostop do pomnilnika RAM in prekinitveni signali, ki smo jih že opisali.

Poglavje 3

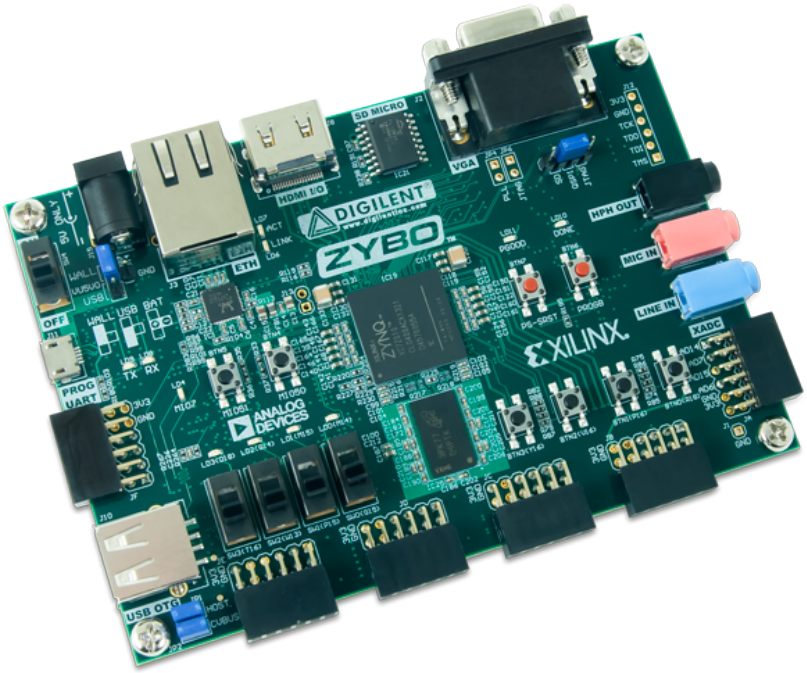
Implementacija aplikacije za procesiranje zvoka

V nadaljevanju bomo predstavili našo implementacijo aplikacije za procesiranje zvoka in ob tem opisali postopke implementacije.

Naš namen je bil izdelati aplikacijo, ki bi bila tipičen primer uporabe programabilnega sistema na čipu Zynq. Odločili smo se za primer digitalnega procesiranja signalov, saj je to dober primer, ki uporablja tako procesorski sistem kot programabilno logiko. Obdelava signalov se dogaja v PL, komunikacija z uporabnikom in vsa vezivna logika pa v PS.

Namen aplikacije je zajem zvoka in določitev osnovne frekvence zvoka, kar dosežemo s hitro Fourierovo transformacijo, ki se izvaja v PL. Zvok se zajema s pomočjo zvočnega kodeka, ki se ga konfigurira po protokolu I2C, zvočni vzorci pa se pretakajo po protokolu I2S. Poleg določanja osnovne frekvence zvoka zna naša aplikacija tudi predvajati posneti zvok in predvajati sintetizirano kitarsko melodijo, ki posnema pravi zvok kitare. To dosežemo s Karplus-Strongovim algoritmom [14].

Jedro implementacije je seveda Zynq, ki pa ga samega ne bi mogli zares uporabljati. Zato uporabimo razvojno ploščico Zybo podjetja Digilent, ki jo vidimo na sliki 3.1. Zybo gosti najmanjši čip iz družine Zynq-7000, Z-7010.



Slika 3.1: Razvojna ploščica Digilent Zybo [18].

Glavne lastnosti Zynqa Z-7010 [18]:

- ARM-ovi jedri tečeta pri 650MHz
- 4400 logičnih rezin
- 240 KB bločnega RAM-a
- 80 rezin za digitalno procesiranje signalov

Poleg Zynqa Zybo vsebuje še množico multimedijskih in komunikacijskih perifernih naprav, zaradi česar je zares uporaben za implementacijo zelo različnih aplikacij. Ploščica je namenjena učenju dela s Zynqom in implementaciji testnih sistemov pred izdelavo končnega izdelka. Glavne Zybove lastnosti so [18]:

- RAM DDR3 s 512 MB in hitrostjo prenosa podatkov 1050 Mb/s

- HDMI z obema načinoma (izvor in ponor slike)
- VGA s 16 biti na piksel
- Ethernet (1Gbit/100Mbit/10Mbit)
- reža in podpora za kartice MicroSD (kamor lahko naložimo OS)
- zunanji pomnilnik EEPROM
- zvočni kodek s tremi vtiči – stereo izhod, stereo vhod line in in mono vhod za mikorofon
- 128 Mb serijskega pomnilnika flash
- možnost programiranja prek JTAG in pretvornik med UART in USB
- 10 stikal in 5 diod LED
- 6 vtičev za razširitvene module

Za nas so torej ključnega pomena RAM, kjer hranimo zajete zvočne vzorce in rezultate FFT-ja, zvočni kodek, ki omogoča zajem in predvajanje zvoka in možnost komunikacije prek UART-a za interakcije z aplikacijo.

Programabilni del Zynqa sprogramiramo s pomočjo Xilinxovega Vivada, razvojnega okolja, ki je naslednik upokojenega okolja ISE. Sami uporabljamo Vivado 2016.2. Vivado je še posebej primeren za delo s Xilinxovimi programabilnimi SoC-i, olajša pa tudi delo na drugih njihovih čipih. Sicer pa se s prenehanjem nadgrajevanja ISE-ja in prihodom Vivada spreminja način dela z FPGA-ji. Zaradi veliko krajšega časa razvoja in večje zanesljivosti se vse več dela z že razvitimi in testiranimi logičnimi bloki *IP – Intelectual Property*, ki jih ponujajo proizvajalci, npr. Xilinx [3]. Tako z namestitvijo Vivada dobimo tudi množico blokov IP, ki so zbrani v *IP-katalogu*.

3.1 Razvoj lastnega logičnega bloka

Xilinx za izdelavo lastnih blokov IP podpira naslednje načine, ki so natančneje opisani v viru [3]:

- **koda v HDL**, iz katere v Vivadovem *IP Packagerju* naredimo kompatibilen logični blok AXI
- **Vivado HLS** (*High-Level Sythesis*), ki kodo v programskem jeziku C, C++ ali SystemC prevede v HDL (VHDL ali Verilog) in tudi že ustvari blok IP z izbranim komunikacijskim vmesikom AXI
- **Matlabov HDL Coder**, ki iz modela v Simulinku zgenerira blok IP
- **System generator**, ki je zelo podoben prejšnjemu načinu, le da lahko pri tem načinu v Simulinkovem modelu uporabimo veliko Xilinxovih blokov, ki so povsem enaki tistim za na FPGA. Za testiranje bloka FFT smo uporabili System Generator, saj smo tako pri določenih vhodih, ki smo jih lahko kontrolirano izbrali, videli točno, kakšni bodo rezulati.

Da hkrati prikažemo še delovanje Vivada, lastni logični blok implementiramo v VHDL-u. Implementiramo generator psevdonaključnih števil, ki smo ga uporabili tudi v končnem sistemu za procesiranju zvoka. Naključna števila rabimo pri že omenjenem Karplus-Strongovem algoritmu. Seveda bi lahko uporabili C-jevo funkcijo `rand()`, a implementiramo svojo v PL zaradi prikaza, kako to storiti.

Psevdonaključni generator temelji na n-bitnem pomikalnem registru, le da je vhod vanj linearna funkcija njegovega prejšnjega stanja [6]. Pri tem je n število bitov naključnega števila. Tak register se imenuje **pomikalni register z linearno povratno vezavo (LFSR)**. Linearna funkcija so navadno ekskluzivne OR operacije na nekaterih bitih. Na katerih bitih se te operacije dogajajo, lahko opišemo s polinomom n-te stopnje, biti pa so določeni z neničelnimi elementi polinoma. LFSR ima najdaljšo periodo ponavljanja izhoda, če je polinom, ki ga opiše, primitiven. Za izvedbo v vezju smo uporabili

Galoisev LFSR [12] s primitivnim polinomom

$$x^{32} + x^{22} + x^2 + x^1 + 1 \tag{3.1}$$

```
architecture Behavioral of random_generator is
    signal rand_temp   : STD_LOGIC_VECTOR(31 downto 0):=
            (0=>'1', others=>'0');
    constant polynomial: STD_LOGIC_VECTOR(31 downto 0):=
            "10000000001000000000000000000011";
begin
    rand_o <= rand_temp;

process(clk_i, rand_temp)
    variable lsb       : STD_LOGIC;
    variable lsb_vector: STD_LOGIC_VECTOR(31 downto 0);
begin
    lsb := rand_temp(0);
    for i in 0 to 31 loop
        lsb_vector(i) := lsb;
    end loop;

    if rising_edge(clk_i) then
        if (resetn_i = '0') then
            rand_temp <= (0=>'1', others=>'0');
        end if;

        rand_temp <= ('0' & rand_temp(31 downto 1)) xor
                     (lsb_vector and polynomial);
    end if;
end process;
```

Izvorna koda 3.1: Implementacija registra LFSR v VHDL-u s fiksno dolžino izhoda 32 bitov, saj moramo ob spreminjanju dolžine spremeniti tudi polinom, kar bi kodo za npr. 30 različnih dolžin občutno podaljšalo.

Naš generator ima 2 vhoda in 1 izhod, kar je razvidno iz kode 3.2

```
Port(
    clk_i     : in  STD_LOGIC;
    resetn_i  : in  STD_LOGIC;
    -- nakljucno stevilo
    rand_o    : out STD_LOGIC_VECTOR(31 downto 0)
);
```

Izvorna koda 3.2: Vhodi in izhodi v modul.

Zdaj ko poznamo celotno HDL-kodo generatorja, lahko pokažemo, kako naredimo logični blok v Vivadu in ga dodamo v IP-katalog.

1. Najprej v Vivadu ustvarimo nov projekt, ga poimenujemo, izberemo lokacijo in sledimo čarovniku. Ko pridemo do okna za dodajanje izvornih datotek, spodaj izberemo ciljni jezik, ki je v našem primeru VHDL. Ko pridemo do zadnjega okna za izbiro čipa oz. razvojne ploščice, izberemo slednjo.

2. Če ne najdemo Zyba, moramo dodati datoteke, ki opisujejo ploščico, med ostale na lokacijo, kjer je Vivado nameščen. Vivado namreč z namestitvijo ne namesti tudi datotek za Zyba. Datoteke dobimo na [17], pod *Design Resources*, kjer izberemo ustrezno datoteko glede na verzijo Vivada. V preneseni datoteki poiščemo mapo `zybo` in jo skopiramo v `\Xilinx\Vivado\2016.2\data\boards\board_parts`.

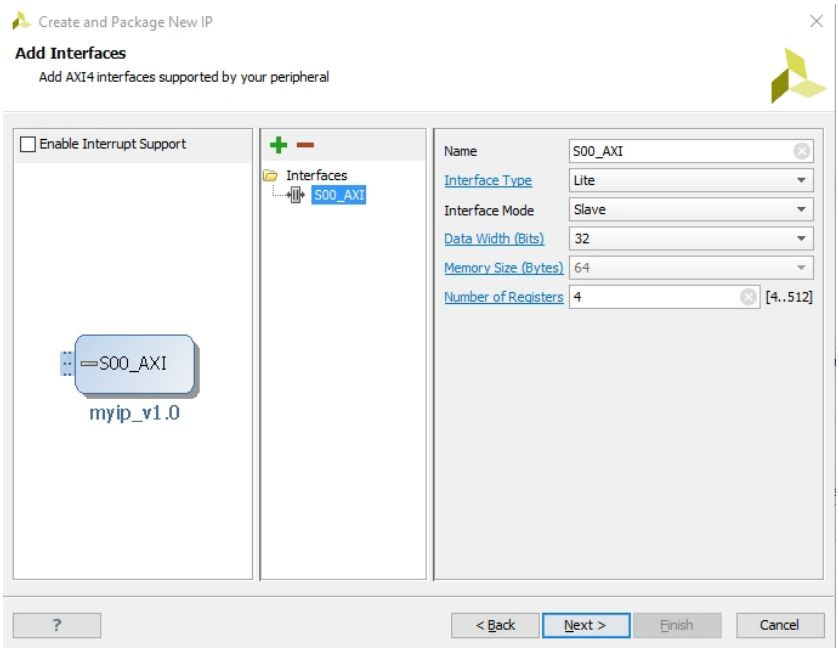
3. Če smo dodali datoteke, moramo Vivado zapreti in ponovno zagnati. Zdaj lahko izberemo Zyba, pritisnemo *next*, preverimo nastavitve in zaključimo.

4. Ustvaril se je nov projekt, v katerem v meniju izberemo *Tools* in potem *Create and Package IP*.

5. Nadaljujemo in izberemo *Create a new AXI4 peripheral*. Čarovnik nas pripelje do okna, v katerem vnesemo ime – vnesemo `random_gene-`

rator, verzijo, opis in lokacijo, kjer želimo imeti shranjene naše lastne bloke IP.

6. V naslednjem oknu, ki ga vidimo na sliki 3.2, izberemo ustrezni komunikacijski vmesnik. Ker je naš generator naključnih števil enostaven blok, ki prenaša le nekaj bajtov hkrati, je AXI4-Lite prava različica. Modul deluje v načinu sužnja, saj sam ne zahteva podatkov, pač pa jih le postavi na vodilo, ko je naslovljen. Za širino podatkovnega vodila izberemo 32 bitov, potrebovali pa bomo le en 32 bitni naslovni prostor, torej en register, a je najmanjša možna izbira 4. Tu gre za pomnilniško preslikane registre, do katerih dostopamo z branjem/pisanjem na osnovni naslov pomnilniško preslikane naprave, ki mu prištejemo odmik registra. Prvi register ima odmik 0, drugi 4, tretji 8 itn.



Slika 3.2: Okno za dodajanje komunikacijskih vmesnikov med ustvarjanjem logičnega bloka.

7. Nadaljujemo, v naslednjem oknu izberemo *Edit IP*, da bomo lahko dodali še datoteko z našim generatorjem. Pritisnemo *Finish*.

8. Odpre se novo okno Vivada, v katerem v zavihku *Sources* z desnim miškinim gumbom kliknemo na *Design Sources* in izberemo Add Sources. Izberemo *Add or create design sources*, nadaljujemo in dodamo datoteko s kodo generatorja.

9. Doda se nova datoteka. V vrhnjem modulu moramo zdaj instanciirati naš naključni generator. Odpremo datoteko `random_generator_v1_0` in pod deklaracijo avtomatsko generirane komponente `random_generator_v1_0_S_AXI` dodamo deklaracijo komponente in en signal (koda 3.3).

```
component random_generator is
    port (
        clk_i       : in    STD_LOGIC;
        resetn_i    : in    STD_LOGIC;
        rand_o      : out   STD_LOGIC_VECTOR(31 downto 0)
    );
end component random_generator;

signal rand_int : STD_LOGIC_VECTOR(31 downto 0);
```

Izvorna koda 3.3: Dodamo komponento random_generator.

10. V avtomatsko generiran modul `S_AXI` dodamo še en nov port, da zgornji del komponente oz. entitete (popravimo v deklaraciji komponente v vrhnjem modulu in v izvorni datoteki) izgleda kot v kodi 3.4 in v modulu popravimo branje iz naslova z odmikom `00` (moralo bi biti v vrstici 350), da je enako kot v kodi 3.5. Tako zdaj namesto iz registra beremo direktno iz izhoda našega generatorja naključnih števil. Vidimo, da v resnici ne potrebujemo nobenega registra, potrebujemo le en naslovni prostor.

```
    port (
    rand_i      : in STD_LOGIC_VECTOR(31 downto 0);
    S_AXI_ACLK  : in std_logic;
```

Izvorna koda 3.4: Dodamo en vhod avtomatsko generiranemu modulu S_AXI.

```
case loc_addr is
    when b"00" =>
        reg_data_out <= rand_i;
```

Izvorna koda 3.5: Popravimo branje z naslova `00` v avtomatsko generiranemu modulu S_AXI.

11. Zdaj pa le še instanciiramo naš generator in povežemo njegov izhod z vhodom v modul `S_AXI` (koda 3.6).

```
random_generator_v1_0_S00_AXI_inst: random_generator_v1_0_S00_AXI
    generic map (
        C_S_AXI_DATA_WIDTH  => C_S00_AXI_DATA_WIDTH,
        C_S_AXI_ADDR_WIDTH  => C_S00_AXI_ADDR_WIDTH
    )
    port map (
        rand_i          => rand_int,
        S_AXI_ACLK      => s00_axi_aclk,

random_generator_inst: random_generator
    port map(
        clk_i       => s00_axi_aclk,
        resetn_i    => s00_axi_aresetn,
        rand_o      => rand_int
    );
```

Izvorna koda 3.6: Povezava vhoda instance S_AXI z izhodom instance random_generator.

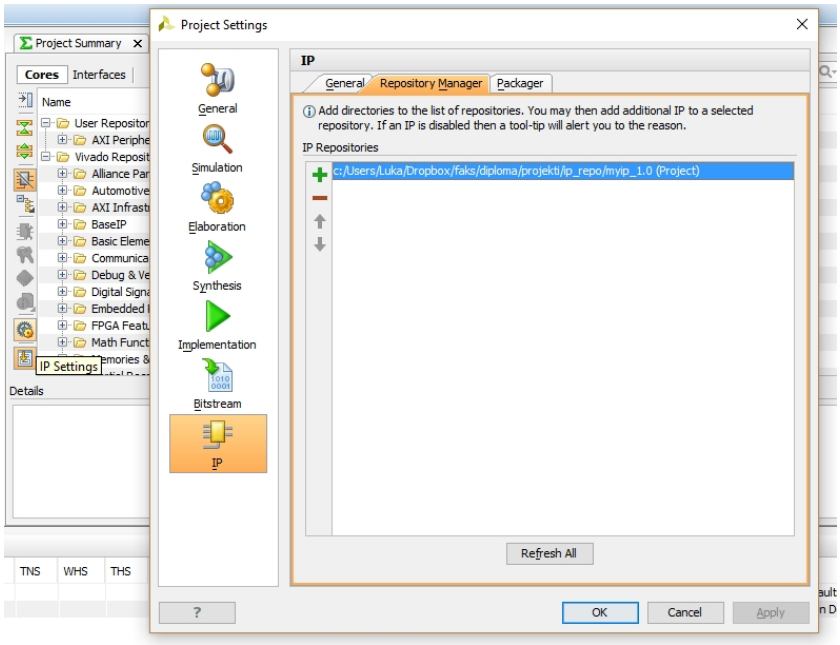
12. Kodiranje je končano, zato v *Flow Navigatorju* na levi izberemo *Package IP* in pregledamo nastavitve. V prvi skupini nastavitev *Identifi-*

cation, nastavimo imena ipd., v *File groups* si lahko ogledamo naše tri datoteke in avtomatsko zgenerirane gonilnike, ki vsebujejo odmike posameznih registrov, test za preverjanje pravilnosti branja in pisanja in makroje za branje in pisanje v registre. V naslednjih skupinah nastavitev bi lahko nastavili še nastavljive parametre – generike, če bi jih imeli (če npr. ne bi generatorja nastavili na fiksno dolžino), v *Customization Parameters* in zmanjšali število naslovljivih pomnilniških mest za vmesnik AXI v *Addressing and Memory*. Izberemo *Review and Package* in nastavimo želene nastavitve, npr. ali želimo, da ta projekt ostane ali se izbriše (ne ustvarjeni IP, le projekt, v katerem smo delali IP), in ali želimo ustvariti arhiv za lažjo prenosljivost bloka IP. Ko izberemo, pritisnemo *Re-package IP*. Ustvarili smo svoj logični blok IP.

Zdaj lahko svoj blok kadarkoli uporabimo v svojem sistemu. Pokazali bomo, kako. Ustvarimo nov projekt ali pa uporabimo že obstoječega, ki bi mu radi dodali generator psevdonaključnih števil. Potem v *Flow Navigatorju* pritisnemo na *IP Catalog* in odpre se nam IP-katalog. V njegovem oknu na levi strani pritisnemo na *IP Settings* (slika 3.3) in izberemo zavihek *Repository Manager*. Pritisnemo na *Add* in dodamo lokacijo, kjer je shranjen IP, ki smo ga ustvarili. Pritisnemo *OK*.

Zdaj, ko smo dodali naš IP-repozitorij v katalog, lahko v bločni shemi (*Block Design*) uporabimo naš IP. Pritisnemo *Create Block Design* v Flow Navigatorju ali pa odpremo bločno shemo, če smo jo ustvarili že prej. V orodnem stolpcu na levi pritisnemo na ikono z zelenim „+“ ali pa pritisnemo *Ctrl+i*. Pojavi se okno z vsemi razpoložljivimi bloki IP, v katerem poiščemo random generator in ga dvokliknemo.

Blok je zdaj v bločni shemi. Nadaljevanje bomo opisali v naslednjih poglavjih. Če bi želeli uporabiti procesorski del, bi morali dodati blok *ZYNQ7 Processing System*, lahko pa uporabljamo tudi samo logični del.



Slika 3.3: IP Catalog in njegove nastavitve.

3.2 Logični bloki, ki smo jih uporabili v implementiranem sistemu

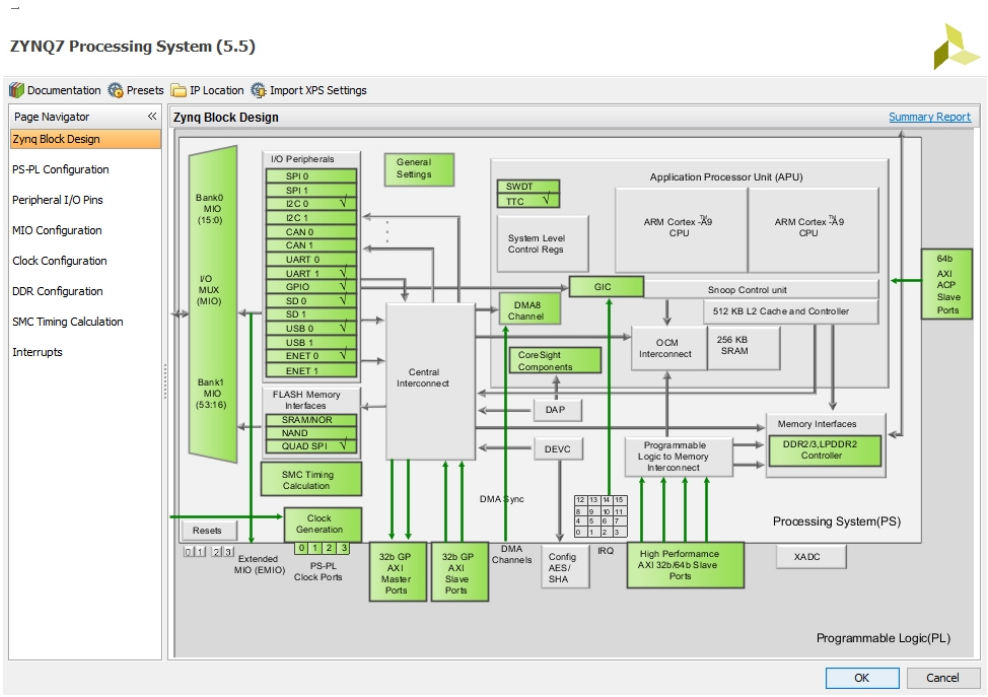
Pri implementaciji sistema smo uporabili več Xilinxovih logičnih blokov in nekaj lastnih in takih razvitih s strani proizvajalcev uporabljenih čipov. Pomembnejše bloke bomo kratko predstavili in pokazali, kako se jih uporablja.

3.2.1 Procesorski sistem ZYNQ7

Procesorski sistem ZYNQ7 – **ZYNQ7 Processing System** je logični blok, ki je pogoj za uporabo procesorskega sistema na čipu. Z njim nastavljamo različne parametre sistema PS, dodeljujemo multipleksirane pine perifernim napravam, omogočamo in onemogočamo njegove vmesnike.

Na sliki 3.4 vidimo grafični vmesnik za njegovo konfiguracijo. Posamezne

nastavitve lahko nastavljamo s klikom na zeleno območje, kar nas postavi v ustrezni zavihek z leve, ki ga sicer lahko odpremo tudi sami.

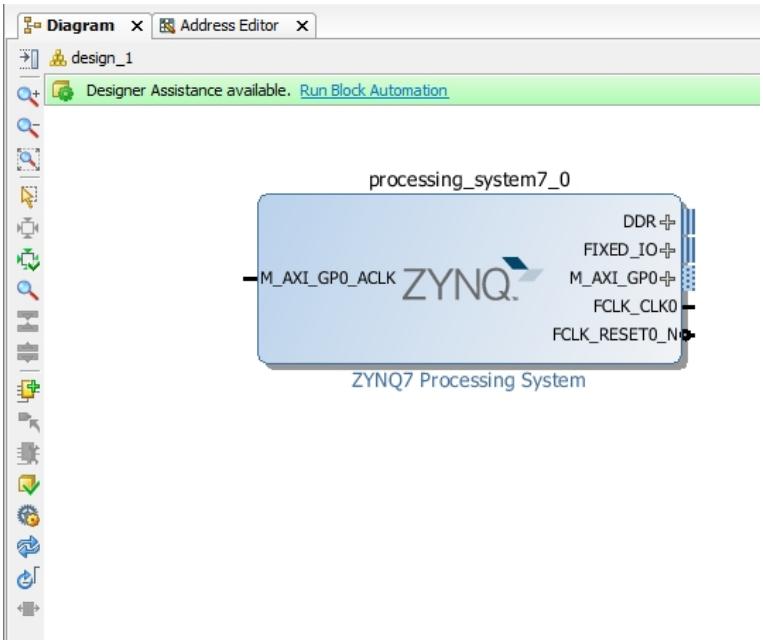


Slika 3.4: Konfiguracija bloka ZYNQ7 PS.

Če uporabljamo razvojno ploščico in v Vivado uvozimo ustrezne datoteke, kar smo opisali v poglavju 3.1, se tiste nastavitve, ki so na razvojni ploščici fiksne, nastavijo avtomatsko, ko zaženemo avtomatizacijo (slika 3.5). Nekatere od teh nastavitev so:

- frekvenca ure, ki pride v čip s ploščice
- konfiguracija pinov na ustrezne periferne naprave, ki so na ploščici fiksno povezane z zunanjostjo, te periferne naprave so npr. pomnilnik Flash, USB, Ethernet, SD, UART, GPIO - te povezave se po avtomatizaciji na shemi pokažejo kot `FIXED_IO`
- nekatere nastavitve perifernih naprav, ki niso povezane s ploščico, a predstavljajo nekakšno inicializacijo sistema, npr. baud rate UART-a

in nastavitev izhoda ure v PL



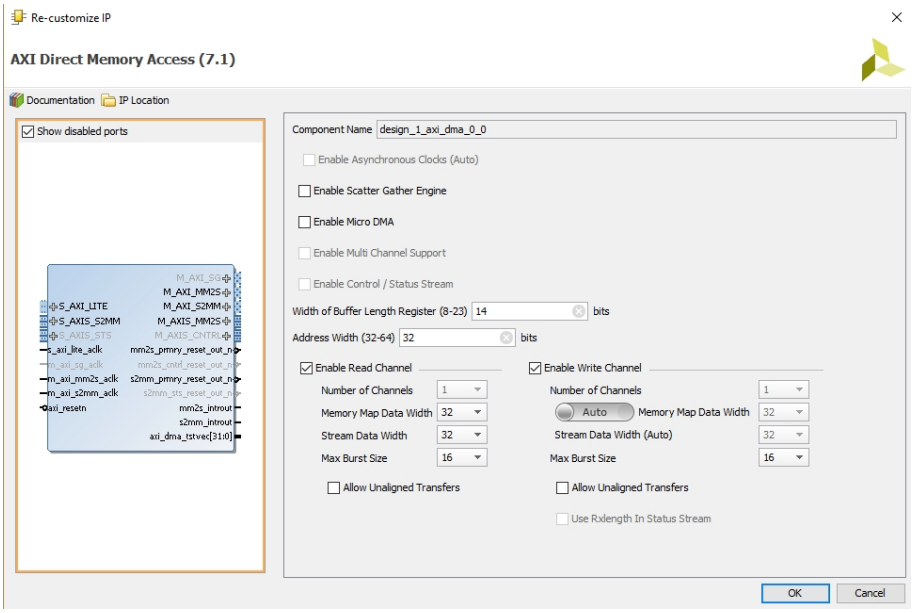
Slika 3.5: Ob pritisku na *Run Block Automation* se avtomatsko nastavijo parametri specifični za razvojno ploščico.

3.2.2 Blok za neposredni pomnilniški dostop (DMA)

DMA – Direct Memory Access je logični blok, ki ima funkcijo neposrednega pomnilniškega dostopa, kar pomeni, da namesto CPE prenaša podatke med V/I-enoto in pomnilnikom RAM. CPE mu mora le naročiti, naj začne prenos, DMA pa ga potem izvede namesto njega. Namen tega pristopa je varčevanje s procesorjevim časom, saj če sam opravlja prenos, medtem ne more početi nič drugega, če pa to zanj počne DMA, lahko v tistem času opravlja katerokoli opravilo.

Še en razlog za uporabo DMA je, da imajo nekateri bloki le vmesnik AXI-Stream, ki pa ga PS nima in zato sploh ne more komunicirati neposredno z njimi. Preslikavo med takim vmesnikom in navadnim pomnilniško preslika-

nim vmesnikom AXI v takem primeru naredi opravlja DMA. Tak blok, ki ima za prenos podatkov le vmesnik AXI-S je tudi XFFT, ki smo ga uporabili.



Slika 3.6: DMA in njegove nastavitve.

Konfiguracijsko okno bloka DMA je na sliki 3.6. Na njej vidimo vse možne nastavitve, ki so natančno opisane v viru [2], mi pa jih povzamemo:

- način **Scatter Gather** na kratko pomeni, da DMA ne potrebuje celotnega sklenjenega bloka v pomnilniku, pač pa zna pisati in brati z več različnih delov pomnilnika. Kazalce na njih hrani v posebnih deskriptorjih. Ta način omogoča največjo hitrost prenosa in najmanj obremeni CPE, a je tudi najbolj prostorsko potraten in najbolj zapleten za uporabo.
- način **Micro DMA** je najbolj prostorsko varčen, a tudi najpočasnejši in je primeren za prenašanje majhne količine podatkov
- **podpora več kanalom** je možna le v načinu Scatter Gather, pomeni pa, da je na DMA lahko priključenih več naprav

- **kontrolni in statusni signali** so prav tako možni le v načinu Scatter Gather
- **velikost registra za dolžino prenosa** je prav to, pri prenosih moramo namreč podati dolžino prenosa, ki je zaradi velikosti tega registra omejena na $2^{23} - 1$
- **velikost naslovnega prostora** pride v poštev pri pomnilniško preslikanih vmesnikih (ki sta dva, eden za pisanje v RAM in eden za branje iz RAM-a)
- **širina podatkov na pomnilniško preslikanem vmesniku**; nastavitev obstaja na obeh kanalih – bralnem **MM2S** – Memory Mapped to Stream in pisalnem – **S2MM** – Stream to Memory Mapped. Z njo povečamo hitrost prenašanja
- **širina podatkov na vmesniku AXI-S** – enako kot prejšnja; je navzgor omejena s prejšnjo
- **največ podatkov v enem prenosu hkrati (burst)** – lahko povečamo hitrost prenašanja
- **poravnava podatkov** – če to funkcijo izklopimo, morajo biti naslovi poravnani

Postopek inicializacije DMA v neposrednem registrskem načinu

Neposredni registrski način (**Direct Register Mode**) pomeni, da ne uporabljamo deskriptorjev za opis podatkov, pač pa naslove in dolžine zapišemo neposredno v registre.

Pri inicializaciji moramo podobno kot pri GIC tudi pri DMA najprej prebrati njegovo konfiguracijo, v kateri je zapisanih večino nastavitev, ki smo jih ravnokar opisali. Te se potem v inicializaciji zapišejo v instanco objekta DMA, ki ga ustvarimo. V inicializaciji se naredi še nekaj preverjanj, DMA pa se na koncu resetira. Če želimo uporabljati prekinitve, jih moramo omogočiti,

za bralni kanal posebej (XAXIDMA_DMA_TO_DEVICE) in za pisalnega posebej (XAXIDMA_DEVICE_TO_DMA). Ali gre za branje ali za pisanje se določi z zornega kota pomnilnika in ne naprave, ki je priključena na DMA. Tudi za DMA Xilinx zagotavlja gonilnike, zato je treba le poklicati nekaj funkcij, kot je razvidno v kodi 3.7.

```
// Look up hardware configuration for device
cfg_ptr = XAxiDma_LookupConfig(dma_device_id);
if (cfg_ptr) {
    // Initialize driver
    status = XAxiDma_CfgInitialize(p_dma_inst, cfg_ptr);
    if (status == XST_SUCCESS) {
        // Enable DMA interrupts
        XAxiDma_IntrEnable(
            p_dma_inst,
            (XAXIDMA_IRQ_IOC_MASK | XAXIDMA_IRQ_ERROR_MASK),
            XAXIDMA_DMA_TO_DEVICE);
        XAxiDma_IntrEnable(
            p_dma_inst,
            (XAXIDMA_IRQ_IOC_MASK | XAXIDMA_IRQ_ERROR_MASK),
            XAXIDMA_DEVICE_TO_DMA);

    return DMA_SUCCESS;
    }
}
```

Izvorna koda 3.7: Inicializacija naprave DMA.

Postopek za začetek prenosa v neposrednem registrskem načinu

Za začetek prenosa v neposrednem registrskem načinu enostavno pokličemo funkcijo, ki preveri, ali je DMA res v tem načinu, ali je prost in ali je naslov ustrezno poravnan oz. ali je vključena funkcionalnost, ki zagotavlja, da to ni potrebno. Potem v register za naslov zapiše naslov, s katerega želimo brati oz. naslov, na katerega želimo pisati, in nazadnje še v register za dolžino zapiše dolžino prenosa, kar tudi sproži prenos.

```
status = XAxiDma_SimpleTransfer(
    p_dma_inst,
    (int)p_rec_buf,
    num_bytes,
    XAXIDMA_DEVICE_TO_DMA);
```

Izvorna koda 3.8: Primer prenosa S2MM.

3.2.3 Blok za hitro Fourierovo transformacijo – FFT

Osrednji del implementiranega sistema je ravno hitra Fourierova transformacija, ki je pravzaprav le pohitrena diskretna Fourierova transformacija [4], ki je definirana kot

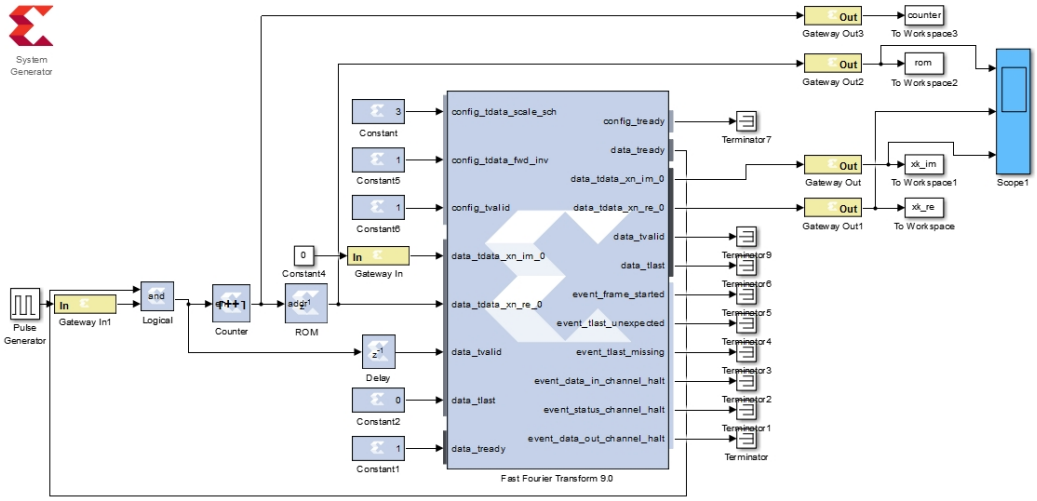
$$X_k = \sum_{n=0}^{N-1} x_n e^{\frac{-2\pi i}{N} kn} \qquad k = 0, ..., N-1 \tag{3.2}$$

Kompleksnost DFT je $\mathcal{O}(n^2)$, FFT pa jo zmanjša na $\mathcal{O}(nlogn)$. Metode za izračun FFT so različne, velja pa, da je FFT vsaka metoda, ki z $nlogn$ operacijami pride do enakih rezultatov kot DFT [11].

Xilinxov blok **XFFT** za izračun DFT uporablja algoritem FFT Cooley-Tukey [15]. Sicer pa je ob načrtovanju sistema v Vivadu moč izbrati štiri različne arhitekture implementacije. Različne arhitekture smo tudi preizkusili v System Generatorju (slika 3.7).

Za naš sistem smo FFT skonfigurirali z naslednjimi nastavitvami:

- 1 kanal z največjo dolžino rezultata omejeno na 16384 točk z omogočeno možnostjo spreminjanja dolžine
- želena frekvenca 100MHz pri cevovodni arhitekturi s neprestanim tokom podatkov (stream)
- podatki so predstavljeni s fiksno vejico in so 32-bitni, 16 bitov za realni del in 16 za imaginarni



Slika 3.7: Model z logičnim blokom *FFT* v Xilinxovem System Generatorju. Uporabljamo ga tako, da *ROM* prednaložimo z vektorjem iz Matlaba, potem pa s števcem naslavljamo zaporedne podatkovne besede, ki gredo na izhod in v *FFT*. Rezultati gredo prek Matlabovega bloka *To Workspace* v delovno okolje v Matlabu, kjer jih lahko izrišemo ali karkoli drugega.

- rezultati posameznih stopenj izračuna se skalirajo, zaokroževanja pa ni, predolgi podatki se porežejo
- vrstni red podatkov na izhodu je naravni, pospeševanje za hitrejše izvajanje v realnem času pa se ne uporablja

Blok XFFT ni pomnilniško preslikan, ima le vmesnike AXI-Stream, zato za spreminjanje njegovih nastavitev uporabljamo blok GPIO (*General Purpose Input/Output*) in enostaven blok lastne izdelave za zaznavo fronte v podatkih, ki pridejo iz GPIO. Razlog, da ga potrebujemo, je, da ima suženjski vmesnik AXI-Stream vhodni signal *s_axis_config_tvalid*, ki smo ga opisali že v 2.3.1. XFFT konfiguracijske podatke sprejme torej šele, ko signal postane 1, pri tem pa zahteva, da to traja le eno urino periodo. Detektor fronte pa ob vsaki spremembi podatkovnega signala *s_axis_config_tvalid* za točno eno

urino periodo postavi na izhod 1, kar lahko vidimo v izvorni kodi 3.9.

```
assign edge_detected = edge_detected_i;

always @ (posedge clk) begin
    tmp <= din;

    if (tmp != din)
        edge_detected_i <= 1'b1;
    else
        edge_detected_i <= 1'b0;
    end
```

Izvorna koda 3.9: Koda detektorja fronte v Verilogu.

Razen nastavitve konfiguracije prvič in vsakič, ko jo spremenimo, ni potrebnega nič drugega. Primer nastavitve je prikazan v kodi 3.10. Spodnjih $log_2 n$ bitov je velikost transforma, potem je en bit, ki pove, ali želimo navadno ali inverzno transformacijo, potem pa še $2 * \text{ceil}(\frac{log_2 n}{2})$ bitov za parametre skaliranja.

```
reg  = (p_fft_inst->scale_sch << FFT_SCALE_SCH_SHIFT) &
                                        FFT_SCALE_SCH_MASK;
reg |= (p_fft_inst->fwd_inv << FFT_FWD_INV_SHIFT) &
                                        FFT_FWD_INV_MASK;
reg |= (int_log2(p_fft_inst->num_pts) << FFT_NUM_PTS_SHIFT) &
                                        FFT_NUM_PTS_MASK;

XGpio_DiscreteWrite(&p_fft_inst->periphs.gpio_inst, 1, reg);
```

Izvorna koda 3.10: Sprememba nastavljivih parametrov bloka XFFT.

Kot že rečeno, ima XFFT le vmesnike AXI-Stream, zato ga moramo za prenos podatkov priklopiti na DMA. Ko želimo, da XFFT izračuna transform, le začnemo prenos podatkov **MM2S** in potem še **S2MM**, da lahko blok takoj, ko izračuna, pošlje podatke prek DMA v RAM. Moramo pa biti pozorni, da oba prenosa zaženemo z enakim številom bajtov, pri tem pa se mora to število ujemati z nastavljeno velikostjo transforma N v XFFT. Če

tega ne upoštevamo, se eden od prenosov ne izvrši do konca (odvisno od tega, ali pošiljamo več ali manj podatkov od N).

3.2.4 Zvočni kodek z vmesnikom I2S

To logično jedro, dostopno na [5], ima nalogo sprejemati zvok od zvočnega kodeka po protokolu I2S in ga pošiljati v pomnilnik in obratno. **I2S** – *Inter-IC Sound* je standard s serijskim vodilom za povezovanje digitalnih zvočnih naprav med sabo. Po njem se med posameznimi vezji prenašajo pulzno modulirani zvočni podatki. Ima ločene signale za uro in podatke, pri čemer ura, ki se imenuje bitna ura, utripne enkrat za vsak diskretni bit podatkov na zaporednem – serijskem podatkovnem vodilu. Tako vidimo, da je produkt frekvence vzorčenja, števila bitov na kanal in števila kanalov, v našem primeru:

$$48000\text{kHz} * 24\text{bit} * 2 = 2,304\text{MHz} \tag{3.3}$$

Vrednost predstavlja minimalno vrednost, ki mora biti zagotovljena, da se ne izgubljajo podatkovni biti. Nič ni narobe, če je večja.
Vodilo je sestavljeno iz vsaj 3 signalov [13]:

- **bitna ura (BCLK)**,
- **ura za izbiro kanala (WS, LRCLK)** – ura s frekvenco enako frekvenci vzorčenja. Ko je v nizkem stanju, se prenaša desni kanal, in levi, ko je v visokem,
- **multipleksiran podatkovni signal**

Prenos zvočnih vzorcev je predstavljen na sliki 3.8.

Logični blok je pomnilniško preslikan, zato lahko do njega dostopamo z branjem/pisanjem na njemu dodeljen naslovni prostor. Tako imamo možnost podatke brati neposredno iz registra, kar pa seveda iz že navedenih razlogov ni najbolje. Ker ima blok tudi vmesnika AXI-Stream, se odločimo za prenos prek njiju. Kot smo že opisali v 3.2.2, za to potrebujemo DMA, ki pa ga

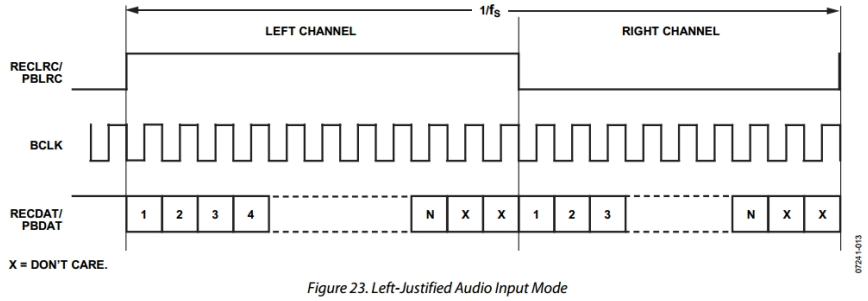


Figure 23. Left-Justified Audio Input Mode

Slika 3.8: Prenos zvočnih vzorcev po standardu I2S [9].

zdaj že znamo uporabljati. Pri implementaciji prenosa podatkov prek AXI-Stream smo imeli nekaj težav, saj je bila v jedru manjša napaka, ki smo jo morali odpraviti. Problem se je kazal tako, da je bila naprava ves čas pripravljena na sprejem (TREADY stalno 1), čeprav je bilo očitno, da se podatki izgubljajo.

Blok ima kar nekaj izhodov povezanih na čipove zunanje pine, saj se pogovarja s kodekom, ki ni na čipu, pač pa na razvojni ploščici. Poleg tega pa kodek zahteva svojo uro, ki mora imeti, če želimo 48-kHz vzorčenje, frekvenco 12.288MHz. Zato moramo iz procesorskega sistema do logičnega bloka I2S speljati še eno počasnejšo uro, kar pa je predstavljalo problem pri sintezi in implementaciji.

Zaradi izhodov krmilnika, ki gredo s čipa in veliko počasnejše ure, moramo ustvariti datoteko omejitev (*constraints)*, ki jo lahko vidimo v 3.11.

```
# Signali I2C iz PS, ki gredo preko EMIO na zunanje pine.
# Potrebni so, ker se kodek nastavlja prek I2C
set_property PACKAGE_PIN N18 [get_ports iic_0_scl_io]
set_property IOSTANDARD LVCMOS33 [get_ports iic_0_scl_io]
set_property PACKAGE_PIN N17 [get_ports iic_0_sda_io]
set_property IOSTANDARD LVCMOS33 [get_ports iic_0_sda_io]

# Usmerimo signale iz I2S na ustrezne pine
set_property PACKAGE_PIN K18 [get_ports {AC_BCLK[0]}]
set_property PACKAGE_PIN T19 [get_ports AC_MCLK]
```

```
set_property PACKAGE_PIN P18 [get_ports AC_MUTE_N]
set_property PACKAGE_PIN L17 [get_ports {AC_PBLRC[0]}]
set_property PACKAGE_PIN M18 [get_ports {AC_RECLRC[0]}]
set_property PACKAGE_PIN M17 [get_ports {AC_SDATA_O[0]}]
set_property PACKAGE_PIN K17 [get_ports {AC_SDATA_I[0]}]
set_property IOSTANDARD LVCMOS33 [get_ports AC*]

# Sintetizatorju povemo, naj logicnih poti od ene ure do druge
# in obratno ne pregleduje (za sinhronizacijo je poskrbljeno)
set_false_path -from [get_clocks clk_fpga_0]
               -to [get_clocks clk_fpga_1]
set_false_path -from [get_clocks clk_fpga_1]
               -to [get_clocks clk_fpga_0]
```

Izvorna koda 3.11: Datoteka `constraints`.

3.3 Prikaz implementacije sistema na PL

V tem poglavju prikažemo, kako smo naš sistem implementirali na programabilni logiki. Prikazali bomo le implementacijo brez XFFT, saj ga je potem, ko se naučimo dodati jedro I2S in ga povezati z DMA, dodati trivialno (dodamo še en DMA in FFT in ju povežemo, kako povezati vhod za konfiguracijo pa smo že opisali). V preteklem poglavju smo opisali vsa logična jedra, ki smo jih uporabljali, nekaj pa se jih zgenerira avtomatsko s pomočjo Vivada.

3.3.1 Procesorski sistem

Najprej torej dodamo procesorski sistem ZYNQ7 in ga ustrezno nastavimo. Pritisnemo *ctrl+i* ali *Add IP* v orodnem stolpcu na levi in poiščemo ZYNQ7 ter ga dvokliknemo. PS je dodan, zato se zgoraj pojavi pomoč načrtovalnega asistenta za povezavo fiksnih povezav, kot smo že opisali v 3.2.1. Kliknemo nanjo in se prepričamo, da je izbrana možnost *Apply Board Preset*. Pritisnemo OK. Zdaj v PS z dvoklikom nanj nastavimo vse, kar bomo potrebovali v nadaljevanju:

- za DMA bomo potrebovali S_AXI_HP (opisano v 2.3.2), zato v PS-PL Configuration odpremo HP Slave AXI Interface in obkljukamo `S AXI HP0 Interface`

- za nastavljanje kodeka bomo na zunanjem pinu potrebovali signal I2C, zato v zavihku MIO Configuration obkljukamo `I2C 0` in se prepričamo, da je v meniju poleg izbran `EMIO`

- za kodek in vmesnik I2S bomo potrebovali tudi počasnejšo uro, zato v zavihku Clock Configuration odpremo PL Fabric Clocks in obkljukamo `FCLK_CLK1`. Frekvenco nastavimo na 12.288MHz, izvor ure pa na ARM PLL, ker s tem dosežemo najbližjo frekvenco želeni

- DMA bo pošiljal zahteve po prekinitvah, zato moramo omogočiti še prekinitve v zavihku Interrupts. Obkljukamo Fabric Interrupts in odpremo PL-PS Interrupts Ports, kjer obkljukamo `IRQ_F2P`

- zdaj le še pošljemo signale I2C oz. IIC na zunanje pine. Na bloku se je pojavila nova oznaka `IIC_0`, na katero kliknemo z desno miškino tipko in izberemo možnost *Make External* (glej sliko 3.9)

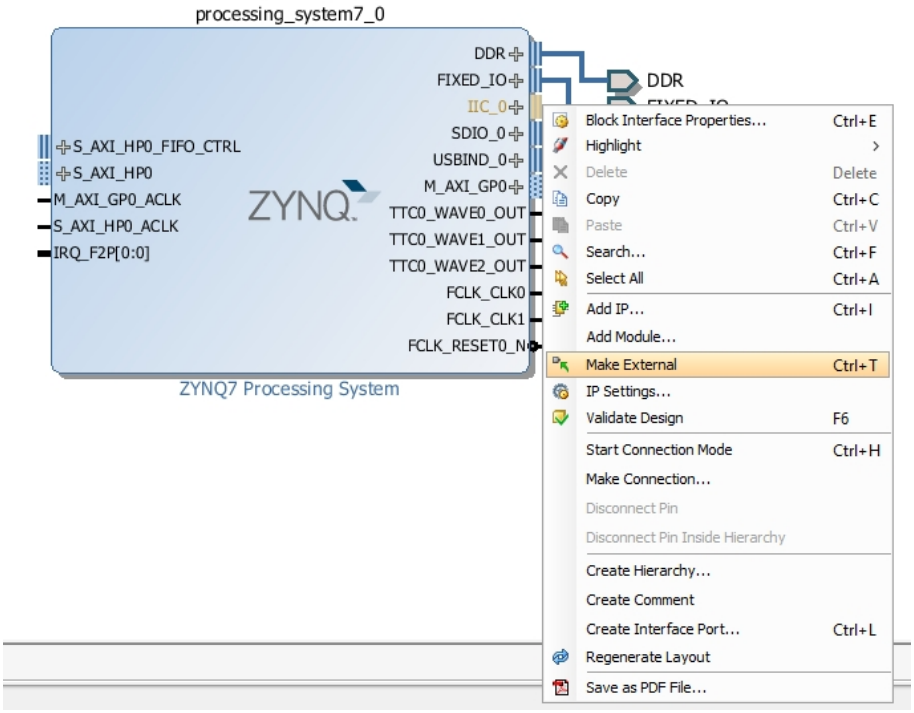
Z nastavljanjem procesorskega sistema smo končali.

3.3.2 Vmesnik I2S

Zdaj iz IP-kataloga dodamo zvočni vmesnik I2S:

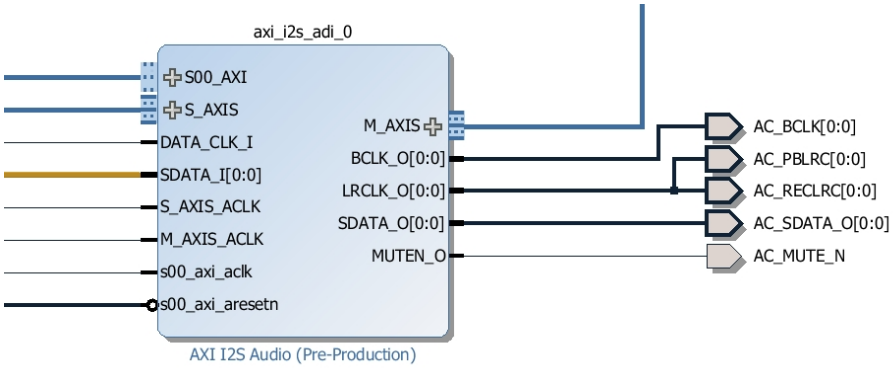
- najprej lahko zaženemo avtomatizacijo, ki bi morala suženjski vmesnik AXI vmesnika I2S povezati z gospodarjem AXI_GP na PS. Kar naenkrat dobimo še 2 novi komponenti: sistem za resetiranje in AXI Interconnect, ki smo ga že opisali v 2.3.2

- z uro `FCLK_CLK1` storimo enako kot prej z IIC in jo pošljemo na zunanji pin (zaradi kodeka)



Slika 3.9: Pošljimo signale IIC na zunanje pine.

- pin `DATA_CLK_I` vmesnika I2S povežemo na to isto uro. To storimo tako, da miško postavimo na pin in ko se spremeni v svinčnik, pritisnemo in povežemo z uro
- na zunanje pine pošljemo vse izhode vmesnika I2S, poleg njih pa še `SDATA_I`. Pri izhodih si pomagajmo s sliko 3.10
- povežimo še uri vmesnikov `M_AXIS` in `S_AXIS` kar z uro nadzornega AXI-ja (`S00_AXI`), torej z uro `FCLK_CLK0`
- nadaljevali bomo lahko šele, ko bomo dodali DMA



Slika 3.10: Povezave krmilnika I2S.

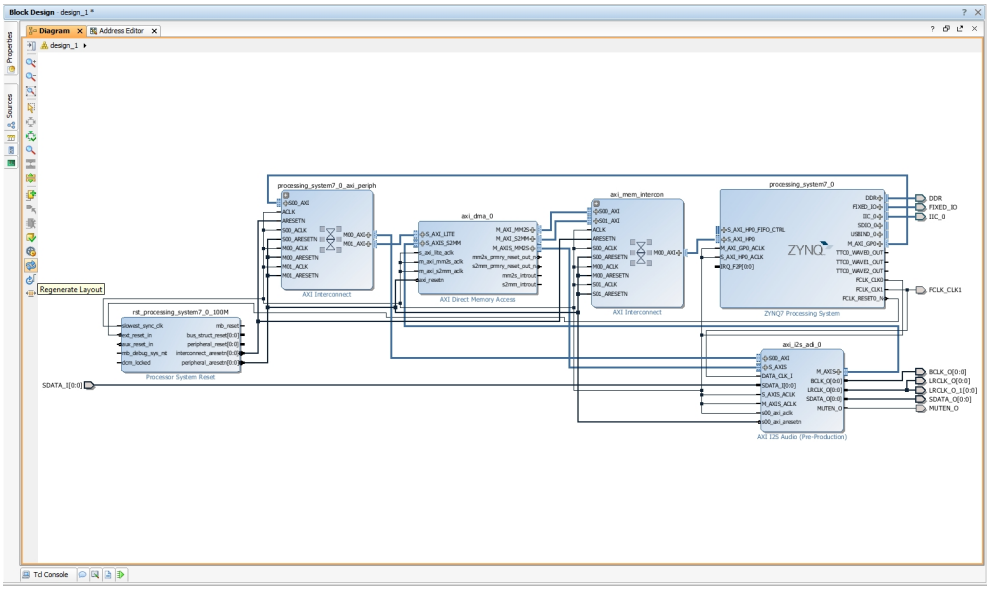
3.3.3 DMA

Iz IP-kataloga dodajmo DMA in ga nastavimo:

- izklopimo način Scatter Gather
- dolžino registra za dolžino nastavimo na 23
- na obeh kanalih nastavimo `Max Burst Size` na 128 in dovolimo neporavnane prenose
- kliknemo OK in zaženemo avtomatizacijo
- obkljukamo vse možnosti in kliknemo OK
- še enkrat zaženemo avtomatizacijo in spet izberemo vse in kliknemo OK. Zdaj smo dobili še en Interconnect, tokrat za visoko zmogljiv AXI-vmesnik na PS.
- zdaj povežemo še kanala AXI-Streaming s krmilnikom I2S – `M_AXIS_MM2S` s `S_AXIS` in `S_AXIS_S2MM` z `M_AXIS`

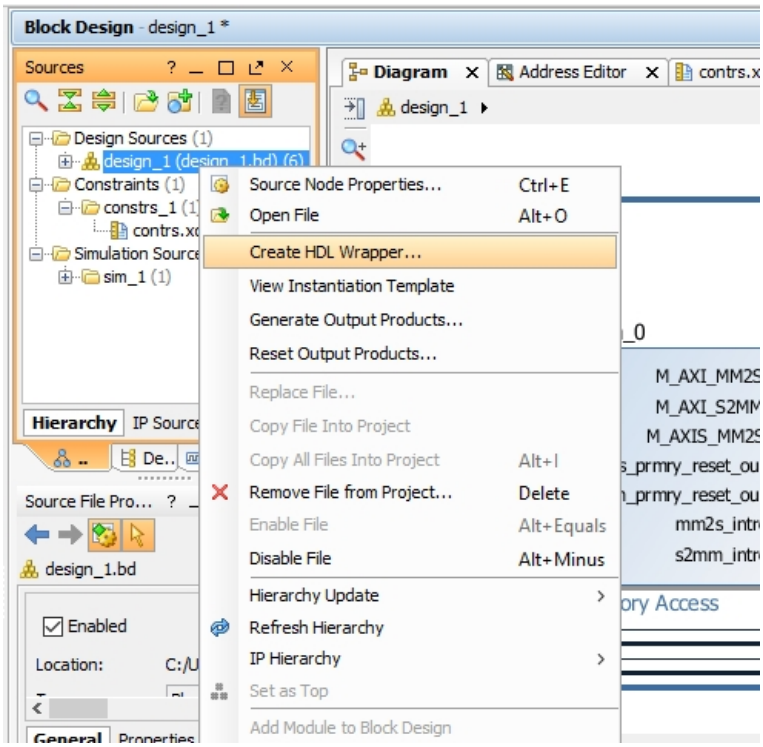
Prišli smo do konca implementacije na PL, dodamo še datoteko *constraints* (3.11), paziti moramo, da se imena ujemajo. Dodamo jo na enako način,

kot smo dodali datoteko generatorja naključnih števil v 3.1, le da tokrat izberemo *Add or create constraints*. Bločna shema bi morala izgledati nekako tako, kot na sliki 3.11. Če ne, lahko izvedemo optimizacijo postavitve s klikom na gumb *Regenerate Layout*, označen na sliki 3.11.



Slika 3.11: Bločna shema sistema.

Zadnji korak je, da zgeneriramo ovojni modul (slika 3.12), ki instanciira bločno shemo, torej cel naš sistem.



Slika 3.12: Zgenerirati je potrebno ovojni modul HDL.

3.4 Kratek uvod v programiranje procesorja

Zdaj smo končali z načrtovanjem sistema v PL, nismo pa ga še zares implementirali. Prej se mora še sintetizirati, zatem implementirati, nazadnje pa se mora zgenerirati datoteka `.bit`, ki predstavlja sliko vezja, ki smo ga ustvarili za programabilno logiko.

V Vivadu v Flow Navigatorju pritisnemo Generate Bitstream in če še nismo izvedli sinteze in implementacije, se bosta izvedli zdaj. Ko se proces konča, v meniju izberemo *File* in bolj na dnu poiščemo meni *Export*, v katerem izberemo *Export Hardware*, pri čemer moramo izbrati *Include bitstream*.

V mapi, v kateri je projekt, se je zdaj ustvarila nova mapa s končnico *.sdk*, ki je namenjena delu v Xilinx SDK – razvojnemu okolju za razvoj programske opreme na Xilinxovih sistemih na čipu. Zdaj ga zaženemo, in sicer tako da izberemo *File* v Vivadu in potem *Launch SDK*.

Odpre se okolje SDK, v katerem v meniju *File* izberemo *New* in *Application project*.

Poimenujemo ga, za *OS platform* pa izberemo standalone, saj ne bo operacijskega sistema. Pod *Target Hardware* je ime mape znotraj prej omenjene mape .sdk. Ta mapa vsebuje datoteko .bit in inicializacijske datoteke. Pustimo kot je in izberemo programski jezik ter možnost, da se ustvari nov podporni paket (*Board Support Package*), ki je zbirka gonilnikov in knjižnic za vse naprave, ki jih uporabljamo. V njem so tudi parametri celotnega sistema (naslovi pomnilniško preslikanih naprav, njihovi ID-ji, ID-ji prekinitev itn.), in sicer v datotekah `xparameters.h` in `xparameters_ps.h`. Pritisnemo *Next* in izberemo predlogo *Hello World*. Pritisnemo *Finish*.

Naredili smo enostaven demonstracijski primer, ki ga lahko zdaj preizkusimo. Zybo mora biti vklopljen v vtič USB računalnika, njegov jumper JP5 pa mora biti nastavljen na `JTAG`. Odpremo aplikacijo za serijsko komunikacijo (npr. *PuTTY* in se povežemo na ustrezni komunikacijski port (preverimo v Upravitelju naprav). Nastavitve so standardne: 115200 bps, 8 podatkovnih bitov, 1 stop bit, brez paritete in brez nadzora pretoka.

V orodni vrstici SDK poiščemo gumb *Program FPGA* (glej sliko 3.13 in ga pritisnemo. Zdaj je FPGA sprogramiran, zatem pa sprogramiramo še procesor z gumbom *Run* (glej sliko 3.13.

V programu za serijsko komunikacijo bi morali videti „Hello World!“.

Uvod v programiranje procesorja na čipu Zynq je zdaj končan.

Slika 3.13: Orodna vrstica okolja Xilinx SDK. Na levi je označen gumb za programiranje FPGA, na desni pa za programiranje procesorja.

3.5 Predstavitev implementirane aplikacije

Nekoliko natančneje bomo predstavili delovanje aplikacije, ki smo jo sprogramirali na sistemu, ki smo ga načrtovali na čipu.

Aplikacija zna torej zajeti zvok iz vhodov za mikrofon in *line in*, a le iz enega naenkrat (kodek ne omogoča zajemanja iz obeh hkrati). Mikrofonski vhod je enokanalni (mono), linijski pa dvokanalni (stereo). Ko je zvok zajet in shranjen v pomnilniku, ga lahko takoj pošljemo nazaj na PL preko vmesnika I2S do kodeka, ki zvok predvaja nazaj.

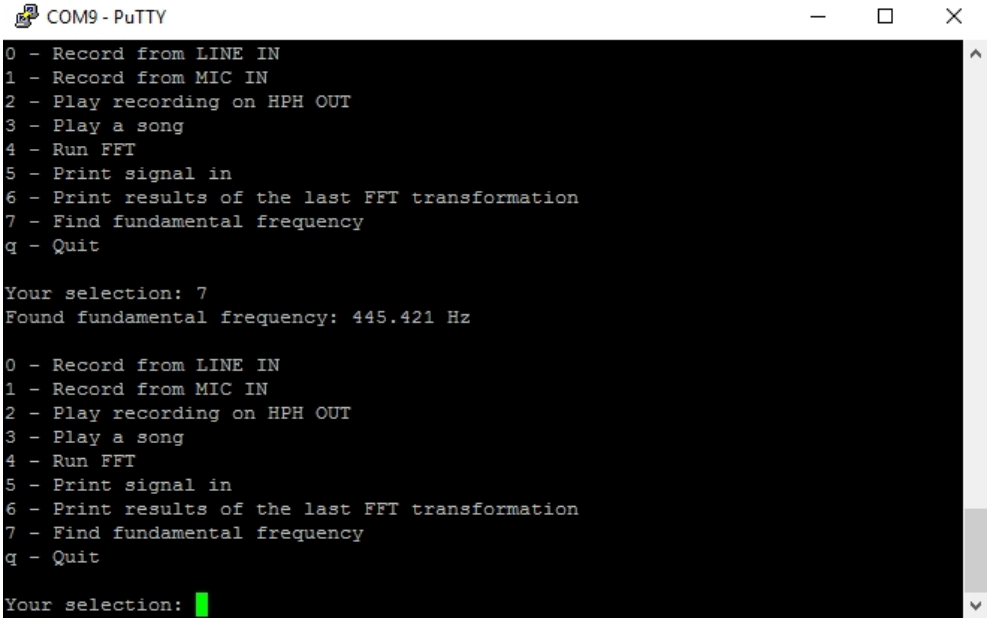
Lahko pa najprej ločimo oba kanala in eno polovico prav tako pošljemo v PL, a ne v I2S vmesnik, pač pa v XFFT, ki vzorce zelo hitro obdela in vrne rezultate v kompleksni ravnini. Nato izračunamo absolutne vrednosti rezultatov po enačbi

$$|z| = \sqrt{\mathrm{Re}^2 + \mathrm{Im}^2} \tag{3.4}$$

Zdaj pa le še poiščemo maksimalno vrednost in ob dovolj lepem tonu dobimo frekvenco tona, ki smo ga posneli. Tu bi morali implementirati nek filter, da bi lahko analizirali tudi signal z nekoliko šuma. Na sliki 3.14 vidimo, kako izgleda interakcija z našo aplikacijo prek PuTTYja.

Druga stvar, ki pa jo še zna naša aplikacija, pa je predvajanje kitarskih tonov. V program smo vnesli tabele z notnimi višinami (za vsako struno posebej smo opisali, katera prečka na kitari je bila prijeta) in notnimi trajanji. Vnesli pa smo tudi frekvence posameznih kitarskih not, tako da lahko potem iz notnih tabel dobimo frekvence.

Ko imamo frekvence, zanje s funkcijo iz kode 3.12 zgeneriramo ustrezno število vzorcev glede na trajanje note.



Slika 3.14: Meni aplikacije. Zgoraj se vidi zadnja najdena osnovna frekvenca 445,421Hz, pri čemer smo posneli ton s 440Hz, kar je dokaj natančno.

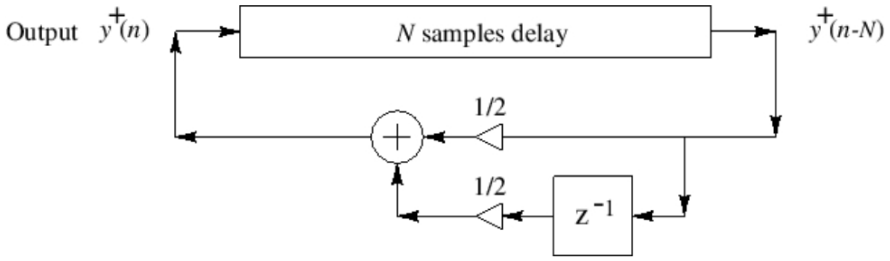
```
u32 n, i;
int delay_buf[num_samp];

// f = fs / (N + d)
// N = fs / f - d
n = SAMPLING_FREQ / freq - 0.5;

int y[num_samp + n];

memset(delay_buf, 0, num_samp*sizeof(int));
for (i = 0; i < n; i++)
    delay_buf[i] = get_rnd_num();

for (i = 0; i < num_samp; i++) {
    // y(n) = x(n) + K*0.5*y(n-N) + K*0.5*y(n-N-1)
    // We ignore the coefficient K, as it is almost 1
    y[i + n] = delay_buf[i] + (y[i+1]>>1) + (y[i]>>1);
```

```
}
memcpy(data_out, y+n, num_samp*sizeof(int));
```

Izvorna koda 3.12: Funkcija za tvorjenje sintetiziranih kitarskih not. Vidi se tudi klic funkcije `get_rnd_num()`, ki vrne naključno število iz našega generatorja naključnih števil na PL.

Funkcija deluje na osnovi Karplus-Strongovega algoritma [14], ki za svoje delovanje potrebuje zakasnitveni medpomnilnik za N vzorcev, množilnik s konstanto K (ki jo mi zanemarimo, ker je med 0,95 in 1) in enostaven nizkoprepustni filter, ki s koeficientom d = 0,5 deluje kot povprečenje (od tod deljenje z 2 oz. pomik za 1 v desno), kar je razvidno s slike 3.15.

$$y(n) = 0{,}5 * x(n) + 0{,}5 * x(n-1).$$



Slika 3.15: Prikaz Karplus-Strongovega modela. Zgoraj je zakasnitveni medpomnilnik, spodaj pa filter [10].

Algoritem deluje tako, da najprej ustvarimo beli šum – N naključnih števil. N se določi glede na frekvenco želeno frekvenco F_0 po enačbi:

$$F_0 = \frac{F_s}{N + d} \tag{3.5}$$

$$N = \frac{F_s}{F_0 - d} \tag{3.6}$$

$$\tag{3.7}$$

Enačbo vidimo tudi v funkciji 3.12. Signal mora biti do želene dolžine napolnjen z ničlami, potem pa ga pošljemo skozi filter, od koder gre na izhod in nazaj v sistem.

Poglavje 4

Zaključek

V delu smo predstavili programabilni sistem na čipu Xilinx Zynq. Predstavili smo njegovo sestavo, tiste elemente, ki smo jih uporabljali sami, pa tudi nekoliko natančneje. V drugem delu pa smo želeli čim bolje pokazati, kako se na čipu načrtuje v praksi. Pokazali smo, kako implementirati različne bloke na programabilni logiki, potem pa še, kako izvoziti sliko vezja in narediti programski projekt.

Lahko bi se bolj osredotočili na implementiran projekt, a se je tudi za nas projekt med delom spremenil v učni primer, saj smo se z Zynqom srečali prvič.

Če bi nadaljevali z delom na projektu, bi morali v programabilno logiko dodati filter za filtriranje signala, saj zdaj analiza deluje le na lepih čistih signalih. Lahko bi tudi odstranili eno napravo DMA in prenose v in iz XFFT in v in iz krmilnika I2S združili v eno.

Za učenje kitare, kot je bilo sprva mišljeno, bi tudi morali imeti nekakšen ekran (prek VGA ali pa HDMI).

Še večje področje, ki bi se ga lahko lotili, pa je, da ne bi programirali procesorskega dela brez OS, pač pa bi dodali Linux ali kak drug operacijski sistem.

Med izdelovanjem diplomske naloge sem se tako zelo veliko naučil, saj se prej nikoli nisem srečal s čipom, ki vsebuje tako FPGA kot procesor. Preden

sem se naloge lotil, sem si predstavljal, da bom veliko več programiral v VHDL, a ni bilo tako. Največ časa sem porabil za učenje o Zynqu in delu z njim, zato tudi v diplomskem delu prevladuje ta vidik.

Literatura

[1] AMBA AXI4-Stream Protocol Specification. Dosegljivo: http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ihi0051a/index.html, 2010. [Dostopano: 13. 9. 2016].

[2] AXI DMA v7.1: LogiCORE IP Product Guide. Dosegljivo: http://www.xilinx.com/support/documentation/ip_documentation/axi_dma/v7_1/pg021_axi_dma.pdf, 2015. [Dostopano: 13. 9. 2016].

[3] Louise Crockett H., Ross Elliot A., Martin Enderwitz A., in Robert Stewart W. *The Zynq Book: Embedded Processing with the ARM Cortex-A9 on the Xilinx Zynq-7000 All Programmable SoC*. Strathclyde Academic Media, 2014.

[4] FMF - Množenje polinomov z metodo FFT. Dosegljivo: http://wiki.fmf.uni-lj.si/wiki/Mnozenje_polinomov_z_metodo_FFT. [Dostopano: 13. 9. 2016].

[5] GitHub - DigilentInc - I2S. Dosegljivo: https://github.com/DigilentInc/vivado-library/tree/master/ip/axi_i2s_adi_1.2. [Dostopano: 13. 9. 2016].

[6] Wikipedia - Linear-feedback shift register. Dosegljivo: https://en.wikipedia.org/wiki/Linear-feedback_shift_register. [Dostopano: 13. 9. 2016].

[7] Wikipedia - Programmable logic array. Dosegljivo: https://en.wikipedia.org/wiki/Programmable_logic_array. [Dostopano: 12. 9. 2016].

[8] Wikipedia - System on a chip. Dosegljivo: https://en.wikipedia.org/wiki/System_on_a_chip. [Dostopano: 12. 9. 2016].

[9] Analog Devices - SSM2603 Data Sheet. Dosegljivo: http://www.analog.com/media/en/technical-documentation/data-sheets/SSM2603.pdf. [Dostopano: 13. 9. 2016].

[10] Stanford - The Karplus-Strong Algorithm. Dosegljivo: https://ccrma.stanford.edu/~jos/pasp/Karplus_Strong_Algorithm.html. [Dostopano: 14. 9. 2016].

[11] Wikipedia - Fast Fourier transform. Dosegljivo: https://en.wikipedia.org/wiki/Fast_Fourier_transform. [Dostopano: 13. 9. 2016].

[12] Wikipedia - Galois Linear-feedback shift register. Dosegljivo: https://en.wikipedia.org/wiki/Linear-feedback_shift_register#Galois_LFSRs. [Dostopano: 13. 9. 2016].

[13] Wikipedia - I2S. Dosegljivo: https://en.wikipedia.org/wiki/I%C2%B2S. [Dostopano: 13. 9. 2016].

[14] Wikipedia - Karplus-Strong string synthesis. Dosegljivo: https://en.wikipedia.org/wiki/Karplus%E2%80%93Strong_string_synthesis. [Dostopano: 13. 9. 2016].

[15] Fast Fourier Transform v9.0: LogiCORE IP Product Guide. Dosegljivo: http://www.xilinx.com/support/documentation/ip_documentation/xfft/v9_0/pg109-xfft.pdf, 2015. [Dostopano: 13. 9. 2016].

[16] AXI Reference Guide. Dosegljivo: http://www.xilinx.com/support/documentation/ip_documentation/ug761_axi_reference_guide.pdf, 2011. [Dostopano: 13. 9. 2016].

[17] Digilent - Zybo. Dosegljivo: https://reference.digilentinc.com/reference/programmable-logic/zybo/start?redirect=1id=zybo:zybo. [Dostopano: 13. 9. 2016].

[18] Digilent - Zybo Reference Manual. Dosegljivo: https://reference.digilentinc.com/reference/programmable-logic/zybo/reference-manual. [Dostopano: 13. 9. 2016].

[19] Zynq-7000 All Programmable SoC: Technical Reference Manual. Dosegljivo: http://www.xilinx.com/support/documentation/user_guides/ug585-Zynq-7000-TRM.pdf, 2015. [Dostopano: 12. 9. 2016].